RADC-TR-69-313, Volume I
Final Technical Report
September 1969

A HANDBOOK ON FILE STRUCTURING

Applied Data Research, Incorporated

This document has been approved
for public release and sale; its
distribution is unlimited.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

181

A HANDBOOK ON FILE STRUCTURING

Robert M. Shapiro
Harry Saint
Robert E. Millstein
Anatol W. Holt
Stephen Warshall
Louis Sempliner

Applied Data Research, Incorporated

FOREWORD

This final technical report was prepared by Messrs. R. M. Shapiro, Harry Saint, R. E. Millstein, A. W. Holt, S. Warshall and L. Sempliner of Applied Data Research, Inc., Corporate Research Center, 450 Seventh Avenue, New York, N.Y. 10001, under Contract F30602-69-C-0034, Project 4594. Contractor's report number is CA-6908-2331.

The Rome Air Development Center project engineer was Miss Patricia Langendorf (EMIDD).

This report consists of two volumes:

Volume I: A Handbook on File Structuring

Volume II: The Representation of Algorithms

This technical report has been reviewed by the Office of Information (EMLS) and is releasable to the Clearinghouse for Federal Scientific and Technical Information.

This technical report has been reviewed and is approved:

Approved: Patricia M. Langendorf
PATRICIA M. LANGENDORF
Project Engineer

Approved: A E Stoll
A. E. STOLL, Colonel, USAF
Chief, Intelligence and Recon Division

FOR THE COMMANDER: Irving J. Gabelman
IRVING J. GABELMAN
Chief, Plans Office

ABSTRACT

This report makes an initial attempt at presenting a coherent approach to the design and analysis of file structures. The relative efficiency of different file implementations is discussed as a function of usage statistics. The fundamental differences between item and descriptor-organized files are discussed in terms of input-output requirements. The report concludes with a discussion of batching, buffering and concurrency.

TABLE OF CONTENTS

VOLUME I: A HANDBOOK ON FILE STRUCTURING

VOLUME II: THE REPRESENTATION OF ALGORITHMS
(separate book)

INTRODUCTION

In this report we make an initial attempt at presenting a coherent approach to the design and analysis of file structures. The work incorporates concepts and techniques developed under Contract AF30(602)-3324 and AF30(602)-4211 and previously described in:

(1) Information System Theory Project: Volume 1, ɒ-Theory. Anatol W. Holt, et al. November 1965. AD 626-819.

(2) Information System Theory Project, The Nature of FFS: An Experiment in ɒ-Theoretic Analysis. The Staff of Project ISTP. March 1966.

(3) Information System Theory Project, Final Report. Anatol W. Holt, et al. September 1968. AD 676-972.

File design at present is a primitive art. Starting with an inadequate definition of the problem, the designer attempts to find an economical representation of the problem on some computing complex. The relative efficiency of different file implementations depends critically upon usage statistics, so that in the course of mapping the problem into the computing milieu, all the usage statistics are in effect assigned values by virtue of the implementation. These implicit and accidental assignments are rarely, if ever, stated. The implemented system levies a cost penalty on all deviations from the implicit statistics. The system rarely

has any capability for collecting the actual statistics in the course of being used; it generally has no facility for adjusting itself to significant deviations.

In document retrieval systems the objective should be to perform all of the functions required by the system, including storage, update, and retrieval, with minimum cost. No meaningful measure of cost can be generated without taking into consideration the frequency of occurrence of each of the functions and the way in which the computing milieu performs these functions. A knowledge of which statistics about the application are worth collecting, and of which characteristics of the representation on the computing milieu are particularly critical or sensitive to the statistics, makes it possible to design a 'system generator' which produces an initial system tailored to what is known _ab initio_ and adaptive in respect to what can be discovered _ex post facto_.

Consider a system consisting of the following four subsystems:

1. A file of documents with each document uniquely named. The functions in this subsystem include:
   - _retrieval_ (input a document name; output a copy of the document);
   - _add_ (input a document; output a document name);

- *delete* (input a document name; optionally output a copy of the document; and in any case prevent the execution of any function which has the document name as an input until the execution of an *add* function with the document name as output);
- *update* (input a document name and a set of revisions; output a document name).

2. A file of descriptors. The functions in this subsystem include:
   - *entry* (input a descriptor; if not previously encountered make up an internal representation; output internal representation);
   - *query* (input a descriptor; output internal representation).
3. A file of descriptor-document relations. The functions in this subsystem include:
   - *query* (input a descriptor [internal representation]; output a set of document names [all of those documents to which this descriptor applies]).
4. User/control subsystem. The functions in this subsystem include:
   - *store* (input a document with descriptors for that document; output a completion signal);
   - *retrieve* (input a list of descriptors combined by *and* and *or* logic; output copies of those

documents which satisfy the retrieval specification).

The focal point of our study will be the organization and utilization of Subsystem 3, the file of descriptor-document relations. We start with an abstract model of cross-indexing (sometimes referred to as coordinate indexing). Feature ard files and edge-notched card files are discussed in relation to this abstraction. Indirect and superimposed coding techniques are explained in this context, and their efficiency is related to usage statistics and hardware characteristics. We then evolve a theoretical measure of the input-output requirements of Subsystem 3, which we characterize as the volume of cross-indexing information. The fundamental differences between item- and descriptor-organized files are discussed in terms of this measure. Formulae are derived for calculating the average volume transacted with, as a function of usage statistics and technique of file organization.

The report then translates the previously developed concepts into the context of computer implementation. A concrete example of the application of the volumetric formulae is provided by a study of PDQ, an information retrieval system operational on IBM System 360 hardware. The effects of batching (a usage statistic) on the formulae are examined. This leads to an important asymmetry from the user's point

of view and suggests a particular design for cross-indexing in an interactive hardware/software milieu. We then present some alternative representational techniques applicable in the computer context. This permits us to derive formulae analogous to the volumetric formulae presented previously and applicable to the three commonly found organizations for cross-indexing files: item-sequenced, inverted, and list-structured files.

We then discuss the problem of performing list intersection, a calculation which is of critical importance in inverted-list file organizations. A new technique for this operation is developed and compared with several existing techniques. Another adaptively applicable representation of inverted lists is discussed. Next, a method for defining graph representations of file structures is presented. The methodology of the report is then applied in a critique of 'balanced trees', a component of the Multilist system. An alternative 'decoding' technique is presented, and use of secondary storage is discussed.

We conclude with a discussion of batching, buffering, and concurrency, explicated by the use of Petri Net models. Both hardware and software models are constructed, including the National Cash Register CRAM card random access memory device and a highly concurrent version of the abstract

cross-indexing model presented in the first section of the report.

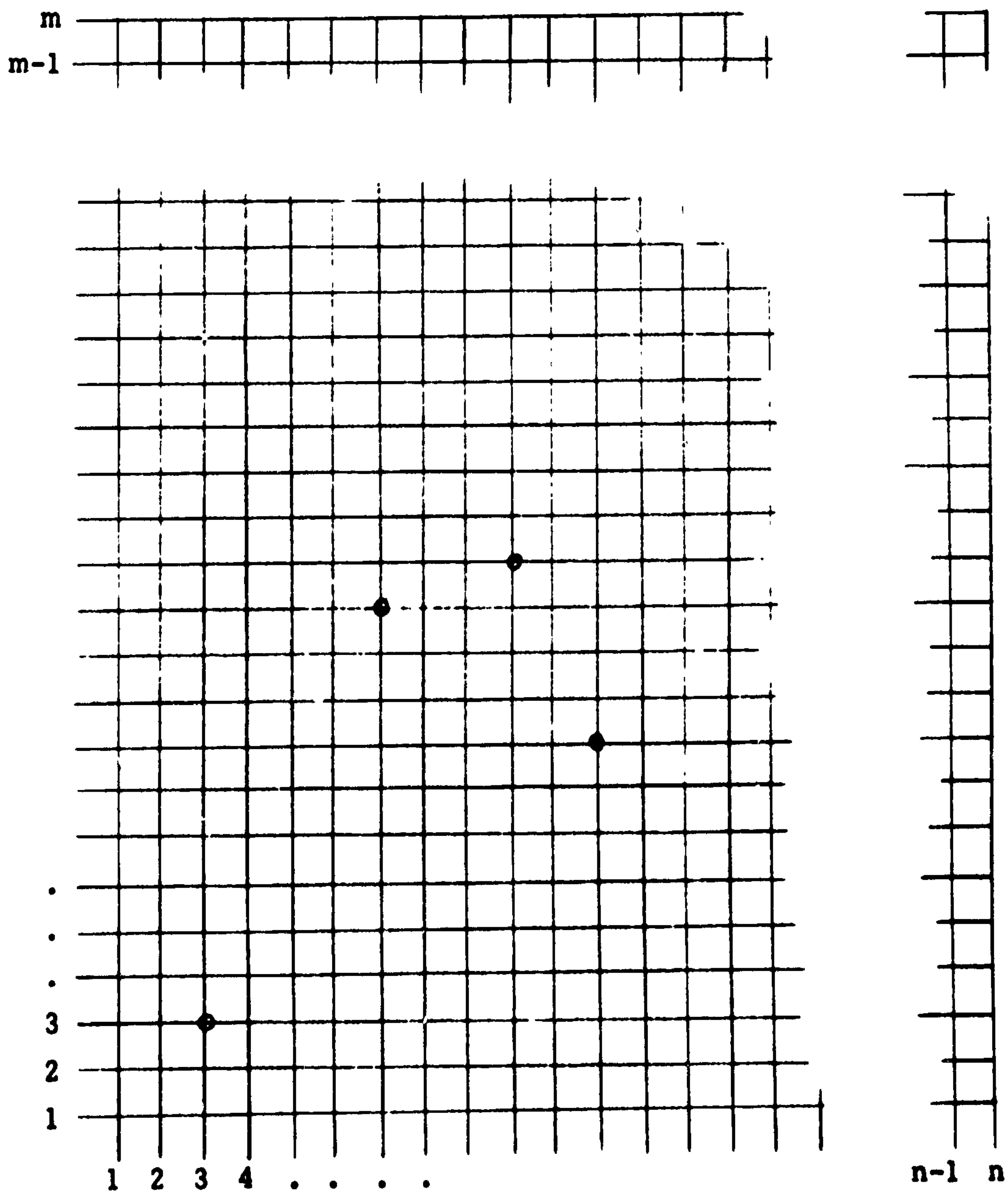
I. A Model of Cross-Indexing

We will use as the basis for our discussion the model of cross-indexing in Figure I-1. This model consists of a grid representing a file of item-descriptor relations. The horizontal lines (rows) are labelled $I_1, I_2, \ldots, I_m$; they correspond uniquely to the items (i.e., documents, records, etc.) currently in the system. The vertical lines (columns) are labelled $d_1, d_2, \ldots, d_n$; they correspond uniquely to the descriptors. Each descriptor applies to some subset of the items in the file, and some subset of the descriptors applies to each item. These relations are represented in the model by circled intersections: each intersection in the grid is either circled or uncircled; a given intersection j,k is circled if and only if descriptor d_j applies to item I_k .

A query or retrieval request consists of a list of descriptors: the response to such a query consists of a list of all items to which all descriptors in the query apply. Thus, in terms of the grid model, a query consists in the selection of some subset Q of the set of all d's. The response consists in a readout of the members of some subset R of the set of all I's. Any given item I_r is a member of R if and only if, for all q such that d_q is a member of Q , q,r is circled.

¶ The operations in this system required for processing queries and updating the file will include: descriptor selection, response readout, circling of intersections, uncircling of intersections, addition of (horizontal and/or vertical) lines, and deletion of (horizontal and/or vertical) lines. We will use this model as a frame of reference for the examination and comparison of various extant indexing techniques. We will begin by considering feature (peekaboo) cards and edge-notched cards, which represent relatively straightforward implementations of two basic types of file organization.

Figure I-1



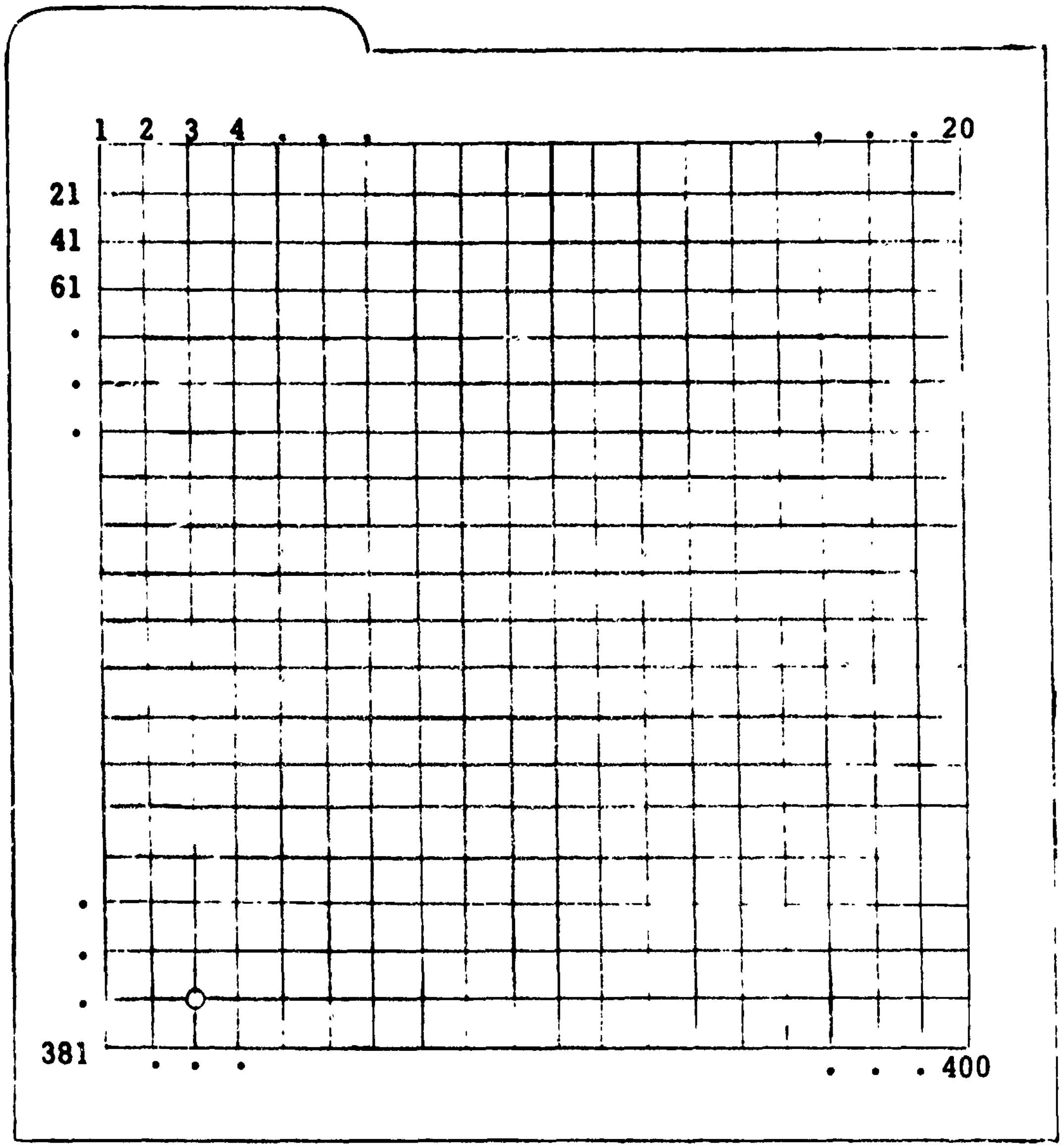
II. Feature Cards

A file of item-descriptor relations may be implemented as a deck of feature cards. In the most straightforward implementation, each card in the deck corresponds uniquely to one descriptor. Each card contains a grid-like array of card-positions, each of which is either punched out or not. (Figure II-1 shows a feature card with 400 positions; they have been numbered from left to right and from top to bottom so that each position is uniquely named.) Each item in the file is assigned a unique card-position -- the same one on every card in the deck. (Thus we could assign item I_k card position k on each card in a deck.) A given position on a given card is punched out if and only if the descriptor represented by that card applies to the item represented by that position. Thus the cards in a deck correspond to the columns in our grid model; the card-positions correspond to the rows; and punched out card-positions correspond to circled intersections.

A query is performed by selecting a subset of the cards in the deck: namely, those cards which represent the descriptors in the query. The selected cards are lined up on top of one another and placed in front of a light source. The positions through which light is visible -- i.e., those positions which are punched out on every card in the query set -- identify the items which satisfy the query.

¶ Note that with feature cards it is relatively easy to add another descriptor to the system -- by simply adding another card to the deck. It is also easy to add another item to the system -- until the number of items is equal to the number of positions on a card. At that point addition of items to the file requires the creation of an additional deck. New items may then be assigned card-positions in the new deck. (If, as in Figure II-1, there are 400 positions on a card and the positions on the first deck have been named by numbering them 1 through 400, then the card-positions of cards in the second deck would be numbered 401 through 800, and so forth.) Thus the number of feature card decks -- and therefore the number of operations necessary to perform one query -- will be equal to: (the number of items)/(the number of card-positions), rounded up. Deletion of an item from the system is relatively difficult since, although it is easy to punch out a hole (i.e., circle an intersection), it is difficult to fill in a hole (i.e., uncircle an intersection). Deletion of an item might involve reproducing -- without the hole in the position representing the item to be deleted -- the feature card for each descriptor which applied to the item.

Figure II-1



III. Edge-Notched Cards

A file of item-descriptor relations may also be implemented as a deck of edge-notched cards. Each card in the deck corresponds uniquely to one item in the system. Each card contains a set of card-positions: a row of holes along its margin. Each hole may be notched out (i.e., the material separating the hole from the edge of the card is cut away) or not. (Figure III-1 shows an edge-notched card with 37 positions, numbered from left to right.) In the most straightforward implementation, each descriptor in the system is assigned a unique card-position -- the same one on every card in the deck. (Thus we could assign descriptor d_k card-position k on each card in a deck.) A given position on a given card is notched out if and only if the descriptor represented by that position applies to the item represented by that card. Thus the cards in a deck correspond to the rows in our grid model; the card-positions correspond to the columns; and notched out card-positions correspond to circled intersections.

A query is made by selecting a subset of the card positions. The deck is lined up and a sorting needle is inserted through each hole which represents a descriptor in the query. The needles are then raised and jiggled so that those cards which have every needled hole notched out drop from the deck. The subset of cards which drop represents the subset

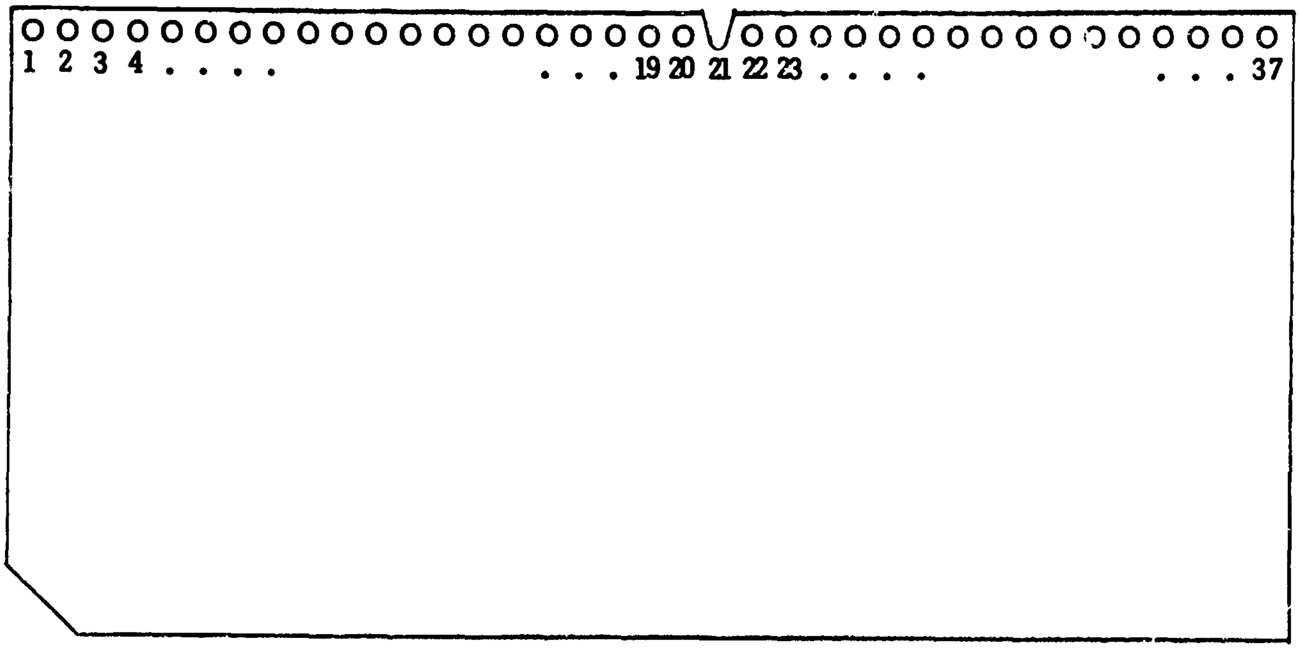
of items to which all descriptors in the query apply.

Note that with edge-notched cards it is relatively easy to add descriptors to the system until the number of descriptors is equal to the number of positions on a card. At that point it becomes difficult to add further descriptors. Starting another deck will not entirely solve the problem since coordinated searches may not be satisfied in either deck by itself.[1] It is easy to add another item to the system -- by simply adding another card to the deck. However, once the number of cards (i.e., items) exceeds the capacity of a sorting needle, the sort operation can no longer be performed in a single step. The number of operations necessary to perform one query will be equal to: (the number of items)/(the capacity of a sorting needle), rounded up. This is, of course, similar to the formula for the number of operations necessary to perform a query with feature cards -- the capacity of a sorting needle corresponds to the number of positions on a feature card. It is, of course, easy to delete an item from the system, in contrast to feature cards. Finally, it is easy to circle an intersection (i.e., notch out a hole) but difficult to uncircle an intersection.

---

[1]An overflow technique is described in ISTP Edge-Notched Card System (A Manual for the Information System Theory Project). Holt, Anatol W. Applied Data Research, Inc. Princeton, N.J. February 1964.

Figure III-1



IV. Indirect Coding

The method of encodement described in the preceding section, in which a card position uniquely represents a single descriptor, is called direct coding. The mechanics of edge-notched sorting and the physical properties of materials for cards, needles, and so forth, set a practical limit to the number of hole positions that can be provided on a margin. (There are a number of different edge-notched card formats available, and the number of positions on a card varies from several dozen to several hundred.) This means that if direct coding is used, the capacity of the card will quickly be exceeded (i.e., the number of descriptors will be greater than the number of card-positions) in all but the most primitive systems. We have already noted that the cost penalty is considerable. Indirect coding is a type of descriptor-encodement which avoids this cost penalty by using a relatively small number of card-positions to represent a relatively large number of descriptors.

The fundamental statistical assumption which justifies indirect coding is the following: although there may be a very large number of different descriptors in a data base, only a very small subset of these descriptors will apply to any one item in the data base. In terms of our grid model, the number of circled intersections on any one

horizontal line will be very small relative to the total number of descriptors in the system.

In order to consider the most straightforward type of indirect coding, let us make a further assumption: the set of descriptors in the system contains subsets of mutually exclusive descriptors -- that is, no two members of the same subset ever apply to the same item. (For example, suppose that the system is a personnel file and each item represents an individual. Since each human being has exactly one birthdate, the age values would form a subset of mutually exclusive descriptors.) Indirect coding can then be utilized as follows: the card-positions are partitioned into subsets. Each such subset of card-positions is used to encode one set of mutually exclusive descriptors.

Suppose, for example, that we wish to represent a set of six mutually exclusive descriptors, which we can name $d_1, d_2 \ldots d_6$, with four-hole positions, named h_1, h_2, h_3, and h_4. We might encode the descriptors as follows:

d_1 : notch out h_1 and h_2
d_2 : notch out h_1 and h_3
d_3 : notch out h_1 and h_4
d_4 : notch out h_2 and h_3

d_5 : notch out h_2 and h_4

d_6 : notch out h_3 and h_4

If d_1 occurred in a query, holes h_1 and h_2 would be needled; if d_2 occurred h_1 and h_3 would be needled, and so forth. The capacity C of a subset of card-positions (i.e., the number of mutually exclusive descriptors encodable in a subset of the holes) is determined by the number of hole positions H in the subset and the number of hole positions P used to encode each of the descriptors.

$$C = H\ (H-1)\ (H-2)\ .\ .\ .\ (H-P+1)/P! = \binom{H}{P}$$

C is a maximum for $P = H/2$, rounded up or down.

This corresponds to a statement in information theory: the encodement which makes the most efficient use of a medium is an encodement which results in a probability of .5 that the value of any given bit will be 1. With edge-notched cards the probability of any given hole being notched out is optimally .5 .

Indirect coding greatly increases the total number of descriptors which can be represented. For example, if we were using an edge-notched card format with 100 holes, direct coding would limit the system to 100 descriptors. However, if we could group the descriptors into 10 subsets

each of whose members were mutually exclusive, then we might represent with the same card format

$10\times C = 10\times 252$ (for $H=10$ and $P=5$) $= 2520$ descriptors

in the system.

This type of indirect coding obviously depends upon the possibility of grouping the descriptors into subsets whose members are mutually exclusive. The effectiveness of the grouping is a function of the number of descriptors in each subset. In the worst case no two descriptors in the data base would be mutually exclusive. Then we would in effect be forced back on direct coding: since the number of subsets would have to equal the number of descriptors, there would only be one descriptor in each group, and consequently we would need one hole position for each descriptor. In fact, in a typical data base some descriptors can be usefully grouped into such subsets while others cannot. In the next section we will discuss another type of indirect coding -- superimposed coding -- which is useful in such situations but which introduces another cost component.

Another difficulty frequently arises in the use of subsets of mutually exclusive descriptors. We often do not have

enough information about the data base to group the descriptors _ab initio_. Nevertheless, with edge-notched cards we must allocate holes when the system is created, and this cannot be done without deciding the number of subsets and the capacity of each. Furthermore, regrouping the descriptors _ex post facto_ requires statistical calculations, the notching out of additional holes, and, worst of all, the filling in of notches. The only alternative to filling in notches is to replace every affected item card with a new, appropriately punched card -- which, of course, will involve redundant notching.

Indirect coding can be employed in our grid model as follows: vertical lines no longer correspond one-one to descriptors; instead, the vertical lines are partitioned into subsets, each of which corresponds to a set of mutually exclusive descriptors, and so forth. This immediately suggests that we can utilize indirect coding with feature cards as well. Since vertical lines in the grid model correspond to cards in a feature card deck, we would proceed as follows: instead of representing each descriptor with one card, we partition the deck into subsets of cards, letting each such subset correspond to one subset of mutually exclusive descriptors, and so forth.

In an edge-notched card system, indirect coding allows

us to represent many more descriptors with a given number of hole positions. Analogously, indirect coding allows us to represent many more descriptors with a given number of feature cards. In an edge-notched system we are in effect forced to use indirect coding since it is impossible to increase the number of hole positions; whereas in a feature card system, indirect coding is used in order to reduce the number of cards required. In both systems the density of circled intersections (i.e., hole punching or notching) is increased. This density approaches an optimum at .5 .

Indirect coding results in an increase in the number of circled intersections (because each instance of a descriptor applying to an item may now be represented by several circled intersections instead of only one). Thus in both edge-notched card and feature card systems we must take into account the increased cost of punching or notching each time a new item is added. Furthermore, when a query is performed, many more vertical lines must be selected: in an edge-notched system more needles must be inserted into the deck; in a feature card system a larger number of cards must be selected and lined up in front of the light source. In other words, indirect coding also leads to certain cost increases for retrieval, and these too must be taken into account. (Later, when we consider

computerized implementations, relative costs will vary significantly from both feature and edge-notched systems, a fact which will help explain why techniques effective in one hardware milieu are totally inappropriate in another hardware milieu.)

The reduction in the number of vertical lines which is made possible by indirect coding has further cost implications. By reducing the number of cards in a feature card deck, we reduce the time required to locate any one card (for example, when performing a query). Analogously, by reducing the number of hole positions used in an edge-notched system, we reduce the time necessary to locate any one particular hole. This cost reduction will be even more pronounced in the computer context. In general, any reduction in memory bulk reduces the amount of time required to locate a particular entry. Therefore, indirect coding will increase the number of vertical lines to be selected (i.e., the number of cards to be selected in a feature card system or the number of holes to be needled in an edge-notched system) in performing a query, but it will also reduce the cost of locating a particular vertical line (feature card, hole) by reducing the total number of vertical lines.

Feature cards are inherently more flexible than edge-notched cards when indirect coding is introduced <u>ex post facto</u>.

The operation is one of replacing a large number of feature cards which are pairwise mutually exclusive (i.e., no two cards have any punched out card-position in common) with a much smaller set of cards. (If n is the number of cards required to represent the set of mutually exclusive descriptors by direct coding, then n', the number of cards necessary with indirect coding, is the smallest integer such that $\binom{n'}{n'/2}$ is larger than n.) All feature cards not included in the subset are unaffected. In an edge-notched system, however, item cards to be replaced may have other descriptors associated with them, and the notches for these descriptors must be replicated on the replacement cards. In other words, the introduction of indirect coding ex post facto will require more circling and uncircling operations in an edge-notched system.

V. Superimposed Coding

Superimposed coding is another type of indirect descriptor encodement which allows the system to have many more descriptors than there are vertical lines. Again it should be true, that although there may be a very large number of descriptors in the data base, only a relatively very small number of the descriptors will apply to any one item. In another respect, however, superimposed coding differs radically from the indirect coding method described in the preceding section: instead of working most effectively when descriptors can be grouped into subsets whose members are mutually exclusive, superimposed coding is most effective when there is no correlation whatever between descriptors -- that is, when probabilities of co-occurrence are entirely random. Superimposed coding is therefore advantageous when nothing is known about the descriptor population ab initio and when the cost of grouping descriptors ex post facto is high. Superimposed coding is clearly advantageous when it is known of the descriptor population that co-occurrence probabilities are random.

Let us first apply superimposed coding to our grid model. We will view all the vertical lines as a single field. Each descriptor is encoded by a small number of vertical lines chosen at random, but such that no two descriptors have the

same code. Suppose, for example, that some descriptor d_i is encoded by vertical lines j, k, and l . If d_i applies to some item I_h , then intersections (j,h) , (k,h) , and (l,h) will be circled.

In an edge-notched implementation, then, all the holes on a card are regarded as a single field. Each descriptor is encoded by notching a small number of hole positions, chosen at random but such that no two descriptors have the same code. Note, however, that two different descriptor codes may have one or more hole positions in common. Moreover, since we have made no assumptions about exclusivity, two descriptor codes may overlap on the same item card. Consequently, superimposed coding creates the possibility of <u>false drops</u> when queries are performed. Suppose, for example, that descriptor d_1 has been encoded by notches at hole positions h_1 and h_{10} , d_2 by notches at h_2 and h_5 , and d_3 by notches at h_5 and h_{10} . Suppose further that d_1 and d_2 both apply to item I_7 but that d_3 does not apply to I_7 . Let us now perform a query containing only the descriptor d_3 . Needles will be inserted through holes h_5 and h_{10} , and the card representing item I_7 will be among those cards which drop from the deck, despite the fact that d_3 does not apply to I_7 .

¶ False drops may be limited by controlling certain statistics when an edge-notched system is designed. The critical factors are: the anticipated maximum number of items in the system, the anticipated number of descriptors applicable to each item, the anticipated number of descriptors in each query, and the cost of handling false drops. An optimal design will guarantee that the average number of notched hole positions per item is less than half the total number of hole positions (again, the probability that a given hole position is notched optimally approaches .5). To satisfy this requirement the following should hold:

$$\bar{d}_I \times P \text{ is less than } .69H ,$$

where $\bar{d}_I$ = the average number of descriptors per item, and P = the number of hole positions used to encode a descriptor, and H = the total number of hole positions. Where this is satisfied, the false drop rate will be less than $.5^q$, where q = the number of hole positions specified by a query.[1] Deviation from these statistics will of course degrade the performance of such a system. If the estimate of the number of descriptors per item is too low, the false drop rate will be higher. (An item card to which too many descriptors

---

[1]See Calvin Mooers, "Zatocoding for Punched Cards", Zator Technical Bulletin 30, 1950. pp. 14-19.

apply will become a "problem card" -- it will have so many holes notched out that it will drop out for virtually every query.) This situation can be dealt with by using overflow techniques.[2] If, on the other hand, the estimate is too high, punching density will be low and space will have been wasted. If the number of descriptors in a typical request is lower than estimated, the false drop rate will increase. It will, of course, also increase if the co-occurrence probabilities for descriptors are not random.

It is extremely important to assess correctly the cost of handling false drops. In an edge-notched system each item card will presumably contain a list of the descriptors which apply to it. The human user can therefore easily sort out the false drops at a cost which may be insignificant relative to the total cost of performing a retrieval. False drops in other systems, such as feature card or computerized systems, may be much more expensive to handle. Let us consider the problem in the context of feature card systems. It is clear that we can translate a grid model system implemented with superimposed coding into an equivalent feature card system. Each descriptor then corresponds to some small subset of feature cards in a deck; the subsets are chosen randomly but in such a way that no two descriptors

---

[2]Holt, op cit.

are represented by the same subset, and so forth. The same statistical criteria for optimal system design will still apply, but the cost factors may be quite different. For example, a "false drop" will mean that when the set of query cards is placed in front of the light source, light shines through some card position representing an item which in fact does not satisfy the query. With edge-notched cards the descriptors could be listed on each item card, but with feature cards the medium does not permit us to write a list of applicable descriptors at each card position. As a result, false drops cannot be detected without going to some other file -- perhaps to the items themselves. (The National Bureau of Standards' microcite system succeeds in avoiding this problem.) This might mean that false drops would not be detected until completion of some relatively expensive operation whose cost is a linear function of the total number of drops, including false drops. In some systems this might be the predominant cost factor.

VI. Combined Coding Techniques

In the preceding sections three coding methods have been discussed: direct coding, indirect coding of sets of mutually exclusive descriptors, and superimposed coding. These three techniques are not mutually exclusive, and in fact, systems have been designed which employ all three.[1] Descriptors which apply to a large fraction of the data base should be coded directly. (For example, if a system included 50 descriptors each of which applied to approximately half of the items, they would co-occur frequently, and on the average 25 of them would apply to each item; consequently any multiple-circling encodement would be extremely inefficient for these descriptors.) Sets of descriptors which are mutually exclusive should be appropriately grouped and coded indirectly. Descriptors which apply to only a small fraction of the data base and whose co-occurrence probabilities are random should be encoded with superimposed coding, provided that the cost of handling false drops is low enough.

---

[1]See, for example, Holt, op cit.

VII. Is Retrieval Time a Linear Function of the Size of the Data Base?

An error frequently encountered in discussions of cross-indexed retrieval operations is the claim that the time required to perform a search in a cross-indexed system does not increase linearly with the number of items in the data base -- whereas in systems which perform an item-sequenced serial search, time is manifestly a linear function of the number of items in the data base, so that:

$$T = C \times I ,$$

where T = the total search time per query, C is some constant, and I = the number of items in the system. In fact, in both edge-notched and feature card systems, search time is a step function of the number of items in the system. We have already expressed this fact in the formulae for the number of operations necessary to perform a query in the two types of system. The critical factors were respectively the capacity of a sorting needle and the number of positions on a feature card. Thus search time in these systems is in fact a step function approximation to $T = C \times I$. Of course, if the capacity of the system is restricted to lie within the first step, it will appear that the time required to perform a retrieval is independent of the number of items in the data base. Naturally, the constant C and the number

of items covered by a step may vary radically from system to system.

An unlikely exception to the above remarks would be the following: a continually growing data base in which the rate of introduction of new descriptors remains constant. Under such circumstances the average number of items to which a descriptor applies would not grow proportionally with the data base. An inverted (or list-structured) file organization implemented on a computer might take advantage of such a situation. Even then, eventually a step function would emerge, but the number of items covered by the first step might be enormous.

One can, of course, prevent retrieval time from increasing proportionally with the size of the data base simply by adding processing capability -- for instance, another person or computer.

VIII. The Volume of Cross-Indexing Information

A cross-indexing retrieval operation involves a number of selections in a hierarchical structure and the sensing, manipulation, and transmission of a volume of bits. Regardless of the particular file-structuring employed, this volume of bits remains constant for a given data base. That is, the raw information content of the volume is simply the total set of item-descriptor relations. There are many ways in which this material may be organized with consequent cost variations. Some methods of organization, when combined with relevant usage statistics, permit further interesting variations on the manner of storage of this volume of bits -- which also result in cost variations. Note that the amount of space used in the host system may vary, but the volume of bits representing cross-indexing information must remain constant.

With edge-notched card systems the volume of bits is represented by the entire card deck. Since each retrieval involves manipulation of the whole deck, edge-notched card retrieval requires transmission of the total volume of cross-indexing information. With feature cards the volume of bits is once again represented by the entire card deck, but each retrieval involves manipulation of only a subset of the deck; namely, those feature cards which formulate the query. In another sense, however, the entire feature

card deck is involved in retrieval: the selection of a particular feature card requires choosing from the total deck the desired card. If the feature cards were organized randomly with no access method other than linear scan, we would have to look at the labels on approximately half the feature cards to locate any particular card. However, the selection of a feature card is, as has been shown previously, analogous to the selection of a hole position in an edge-notched deck.

To compare usefully the volume of bits transacted with in the two systems, we will therefore view the performance of a query as a two-stage process: (1) the selection of the subset of vertical lines (in the grid model) which represents the query; (2) the selection of the subset of horizontal lines which satisfy the query (i.e., those lines which are circled at every intersection with a member of the query set). With respect to the first stage -- selection of a subset of the cards in the feature card system, or selection of a subset of the hole positions in the edge-notched system -- the two systems are equivalent. With respect to the second stage, however, they are not: in the edge-notched system we must still transact with the total volume of bits (i.e., with the entire deck); in the feature card system we need only deal with a subset of those bits (i.e., with the query cards).

IX. A Fundamental Difference between Item- and Descriptor-Organized Files

Let us expand our comparison of edge-notched card systems and feature card systems to a more general comparison of two basic types of cross-indexed file organization. Those files which, like edge-notched systems, are sequenced by item we call item-organized files; files which, like feature card systems, are sequenced by descriptor we call descriptor-organized (or inverted) files.

In general, to perform a retrieval in an item-organized file the entire volume of item-descriptor information must be transacted with; whereas in a descriptor-organized file only a subvolume of the item-descriptor information need be transacted with. Needless to say, the cost significance of this fact is highly dependent upon the interaction between usage statistics, hardware characteristics, and representational technique. For the sake of illustration, consider the following possibilities:

(1) A retrieval request which includes a large proportion of the descriptor population. In this situation the volumetric reduction is negligible. Fortunately, for a large spectrum of information retrieval problems, only a tiny fraction of the descriptor set applies to any one item. Hence a retrieval request which included a large number of

descriptors would not be satisfied by any item in the collection. Therefore, within this spectrum retrieval requests will,on the average, include only a small fraction of the descriptor population. However, when in the sequel we discuss batching, this situation will once again be of interest.

(2) A retrieval request to which a large proportion of the items apply. In this situation the volumetric reduction in the cross-indexing operation is significant, but the size of the response implies that at some stage we will be dealing with most of the items anyway. Under such circumstances that aspect of the indexing operation which involves the descriptor-item pairs is likely to be almost insignificant in terms of the total cost of the operation, and other factors which may or may not depend upon the inverted organization, may dominate the cost. Once again, for a large spectrum of information retrieval problems (but not quite so large as that cited in the previous paragraph) the number of items satisfying a request will, on the average, be quite small relative to the total item population. However, when in the sequel we discuss batching, this situation too will once again be of interest.

X. A Second Fundamental Difference between Item- and Descriptor-Organized Files

Given the symmetry between items and descriptors suggested by our grid model, the fact that the inverted file organization facilitates retrieval would imply that some other operation should be facilitated in an item-organized file. Queries are stated in terms of descriptors and partition the indexing information by descriptor. Loosely speaking, that is why the inverted organization is potentially advantageous in performing retrievals. Additions and deletions of items, on the other hand, group the indexing information by item; here we can expect some advantage from an item-organized file.

What is at issue is the volume of bits that must be transacted with. In an item-organized file, to add or delete an item (i.e., to perform an update) we must manipulate a volume which is $1/I$ of the total volume (where I = the total number of items in the system). In an inverted organization, we must in some sense manipulate d/D of the total volume (where d = the number of descriptors that apply to the item, and D = the total number of descriptors in the system). Since for a wide spectrum of systems the number of items is much larger than the number of descriptors, and d is greater than or equal to one, an update in this spectrum involves a smaller subvolume of

the item-descriptor information if the file organization is by item rather than by descriptor.

Therefore, in the absence of batching (which we will discuss in the sequel), and without introducing the critical effects of usage statistics and hardware characteristics (except on a very general level), we can make some comparative remarks about the volume of cross-indexing information which must be handled to perform retrievals and updates in the two types of file organization.

XI. Formulae for the Volume of Bits Transacted with

(a) $$V = I \times D$$

where V is the total volume in bits

I is the number of items

D is the number of descriptors

(b) $$V_{rD} = I \times \bar{d}_r = (\bar{d}_r/D)V$$

where V_{rD} is the average subvolume for retrieval in a descriptor-organized file

$\bar{d}_r$ is the average number of descriptors in a retrieval request

(c) $$V_{rI} = V$$

where V_{rI} is the average subvolume for retrieval in an item-organized file

(d) $$V_{uD} = (\bar{d}_i/D)V$$

where V_{uD} is the average subvolume for update in a descriptor-organized file

$\bar{d}_i$ is the average number of descriptors for an item

(e) $$V_{uI} = (1/I)V = D$$

where V_{uI} is the average subvolume for update in an item-organized file

(f) $$V_D = pV_{rD} + (1-p)V_{uD} = I(p(\bar{d}_r-\bar{d}_i) + \bar{d}_i)$$

where V_D is the average subvolume for a negotiation in a descriptor-organized file

p is the fraction of negotiations that are retrievals

(g) $$V_I = pV_{rI} + (1-p)V_{uI} = V(p(1-1/I) + 1/I)$$

where V_I is the average subvolume for a negotiation in an item-organized file

XII. Some Comments on the Volumetric Formulae

These results must be interpreted with care, since hardware costs have not been introduced. For instance, when updating a feature card it is perfectly true that a large number of bits are being dealt with (the set of positions on a feature card) but the cheapness and density of the medium may make this operation inexpensive. The relative volumes are not real cost factors, but only a theoretical measure of the amount of information to be transacted with. These results become more interesting when we move from edge-notched card systems and feature card systems (two different hardware milieus, with different costs for transacting with the same volume of information) to a digital computer capable of employing either file organization and using the same hardware to transact with the bit volume, so that the theoretical measure is converted into a useful comparison.

XIII. Computer Implementations of the Cross-Indexing Operation

At this point it should be intuitively clear that the grid model provides a general framework for discussing cross-indexing. We have so far discussed two general classes of extant systems, based on feature cards and edge-notched cards. Both of these use cross-indexing, and we have established the relationship between each of them and the grid model. We will now do the same for a third class of systems, based on the use of digital computers. Because the characteristics of stored-program digital computers permit flexible utilization of hardware resources, the number of significantly different and interesting representational techniques for cross-indexing is far greater than in either edge-notched card or feature card systems, where a combination of direct, indirect, and superimposed coding techniques virtually exhausts the possibilities. Hence our discussion of computer-based techniques is of necessity incomplete; we will concentrate on highlighting the similarities and differences between a few computer techniques and the systems already discussed.

XIV. Computer-Implemented Inverted File Organization

A straightforward computer implementation of an inverted file would consist of a set of lists, one for each descriptor in the system. There are a number of possible organizational schemes for the lists themselves, the following being the simplest: the bits in each list are numbered sequentially; bit j in any given list refers to item I_j . Hence each list is exactly I bits long (where I = the number of items in the system). Each bit has a value of either 0 or 1 . A given bit in a given list is a 1 if and only if the descriptor associated with that list applies to the item associated with that bit position in the list. The lists, then, correspond to the vertical lines in our grid model; bit positions in the lists correspond to horizontal lines; and bits with value 1 correspond to circled intersections.

A retrieval request is made by selecting a subset of the lists and intersecting them. Specifically, the lists which correspond to the descriptors in the query are selected and from these a response list is calculated as follows: a given bit j in the response list has value 1 if and only if bit j of every query list has value 1 . The bits with value 1 in the response list identify the items which satisfy the query. Addition of a descriptor

to the system is accomplished by creating a new list. The addition of an item requires lengthening every list by one bit. An intersection is circled by setting a bit to 1 ; an intersection is uncircled by setting a bit to 0 . The cost of performing these operations may vary enormously as a function of characteristics of the memory medium. If the lists are stored in a bit-addressable ferrite core memory, the cost of changing the value of a bit will be trivial. If the lists are stored on magnetic tape, it may be necessary to read and write all the lists in order to change the value of a single bit.

Such a computer-implemented inverted file is of course similar to a feature card system employing direct coding. Indexing information is grouped by descriptor and not by item. Position within a group (i.e., card-position on a feature card or bit position in a list) identifies the item to which a descriptor applies. Consequently, most of our conclusions about the comparative ease or difficulty of various operations in feature card systems are valid for computer-implemented inverted file systems as well. However, the relative costs may be quite different and will vary enormously according to the details of a computer implementation of the inverted organization. An interesting difference between the two systems appears in the addition of items. With feature cards items can be added to the

system relatively easily until the number of items equals the number of positions on a card. As we have seen, this leads to a step function for the evaluation of retrieval cost relative to the size of the data base. This function will apparently be linear in a computer-implemented inverted file system, but limitations on the amount which can be read in a single transmission can reintroduce a step function.

XV. Computer-Implemented Item-Sequenced File Organization

A straightforward computer implementation of an item-sequenced file would consist of a set of item keys, one for each item in the system. Each item key consists of a set of bits, numbered sequentially; bit position j in any given item key refers to descriptor d_j . Hence each item key is D bits long (where D = the number of descriptors in the system). Each bit has a value of either 0 or 1 . A given bit position in a given item key is a 1 if and only if the descriptor associated with that bit position applies to the item associated with that item key. The item keys, then, correspond to the horizontal lines in our grid model; bit positions in the item keys correspond to vertical lines; and bits with value 1 correspond to circled intersections.

A retrieval request is made by selecting a subset of the bit positions and passing through the entire file, testing each item key for bit inclusion. Specifically, a query key word is constructed: it consists also of D bits numbered sequentially; a given bit position j is a 1 if and only if descriptor d_j is in the query. The query key word is then used to evaluate each item key in the file. If a given item key has a 1 at every bit position at which the query key has a 1 , then the

corresponding item satisfies the query. Addition of an item to the system is accomplished by creating a new item key. The addition of a descriptor requires lengthening every item key by one bit. Circling and uncircling of intersections is accomplished by setting bits to 1 or 0 . As with inverted file systems, the cost of performing these operations will vary enormously depending on characteristics of the memory medium.

Such a computer-implemented item-sequenced file is, of course, similar to an edge-notched card system employing direct coding. Indexing information is grouped by item and not by descriptor. Position within a group (hole-position on an edge-notched card or bit position in an item key) identifies a descriptor which applies to an item. Consequently most of our conclusions about the comparative ease or difficulty of various operations in edge-notched systems are valid for computer-implemented item-sequenced file systems as well, although relative costs may be quite different and will vary enormously, according to details of a computer implementation of the item-sequenced organization. Again, an interesting difference appears in the addition of items. In edge-notched systems adding an item has no effect on retrieval cost until the number of items exceeds the capacity of a sorting needle, which leads to a step function for the

evaluation of retrieval cost relative to the size of the data base. This function will apparently be linear in a computer-implemented item-sequenced file system, but limitations on the amount which can be read in a single transmission can reintroduce a step function.

XVI. The Use of Indirect and Superimposed Coding in Computer Implementations

Having maintained the analogy between our grid model and the computer-implemented file organizations, we can simply state without further elaboration that both indirect and superimposed coding techniques are applicable to computer implementations of both the inverted file organization and the item-sequenced file organization. Clearly, the same statistical conditions required for the use of these techniques in edge-notched and feature card systems will have to hold in order to justify their use in computer-implemented systems. If these conditions hold, then fewer lists will be required in the inverted file organization and fewer bits per item key in the item-sequenced organization.

XVII. PDQ (Program for Descriptor Query)

PDQ is a program implemented on IBM System 360 equipment (with disk packs as the secondary storage medium) which provides an information retrieval capability to the system user. Cross-indexing is provided by an item-organized file of key words. The design principles for the coding scheme are virtually identical to the design principles in edge-notched card systems, and, in fact, PDQ is virtually a computerized version of Anatol Holt's Information System Theory Project edge-notched card design, with some added capability and adaptive features provided by the computer.

The key words are a composite of direct, indirect, and superimposed codes. There is one key word per item. A search is performed by constructing a query key word and performing a bit inclusion test of the query key against every item key. Query and update cost functions are analogous to those for an edge-notched card system -- with the difference being that in edge-notched card systems the cost of retrieval, relative to the number of items, is a step function, whereas in PDQ it is linear.

As we have seen already from our grid model and numerous examples, the coding technique is not determined by the

file organization. The same coding technique could be employed in a $\overline{\text{PDQ}}$ which used an inverted file organization. In both systems, the volume of cross-indexing information is identical. The only difference is in how the bits are grouped together from the point of view of accessing. Referring back to our abstract grid model, it is a question of whether the intersections are stored row-wise (item-oriented) or column-wise (descriptor-oriented). Even if they are pseudo-random access, most computer memories and secondary storage devices require a one-dimensional addressing scheme; the cost of accessing material along another axis is much greater. Hence we must choose either a column or a row grouping. The volumetric formulae presented in Section XI provide criteria for making this decision. The calculations below are based on PDQ figures and compare the two organizations as a function of the ratio of retrievals to updates.

(a) $V_D = I(p(\bar{d}_r - \bar{d}_i) + \bar{d}_i)$ [from Section XI, formula (f)]

(b) $V_I = V(p(1-1/I) + 1/I)$ [from Section XI, formula (g)]

when $V_D/V_I = 1$ the volumetric efficiencies are identical

¶ Assuming the coding technique is effective, the density of 1 bits will be $\frac{1}{2}$ and $\bar{d}_i = D/2$. Substituting and solving for P yields

(c) $$p = \frac{D(I-2)}{3DI-2I\bar{d}_r-2D}$$

since $0 < \bar{d}_r \leq \bar{d}_i = D/2$

(d) $$\frac{I-2}{3I-2} < p < \frac{I-2}{2I-2}$$

and since $I >> 2$

(e) $$\frac{1}{3} \stackrel{\sim}{<} p < \frac{1}{2}$$

which shows that if update transactions comprise more than $\frac{2}{3}$ of the transactions, the item-sequenced organization is superior; if retrieval transactions comprise more than $\frac{1}{2}$ of the transactions, the inverted organization is superior.

These calculations are meaningful but do not tell the whole story, since volumetric considerations are not the only factor. In particular, the following major consideration has been overlooked:

Choosing a feature card is like choosing a hole position. Analogous operations exist in PDQ and the hypothetical $\overline{\text{PDQ}}$.

Both involve some dictionary which converts a descriptor into a bit pattern for inclusion testing or into the address of an inverted list. The difference lies in the fact that the cost of actually obtaining the appropriate inverted lists involves more than simply bit transmission (i.e., of the bit volume already discussed). We must locate the lists in a pseudo-random access memory. Because the set of lists for each retrieval request is different, there can be no way of avoiding an additional cost factor: seek time for each inverted list. As the number of items in the system increases, the number of inverted lists involved in a query or an update clearly does not increase at the same rate as does the volume of cross-indexing bits. Therefore, as the number of items in the data base becomes large, the volumetric considerations will dominate. PDQ is oriented toward a small data base with frequent updates. $\check{P}\overline{DQ}$ would be more suitable for a larger data base with infrequent updates.

XVIII. Batching or Buffering

Thus far we have restricted our attention to retrieval situations which do not permit batching. By batching we mean for instance: the accumulation of a number of queries (or updates) which are then all processed together -- or the concurrent processing of a number of queries. From one perspective, there is no need to distinguish between batching and buffering. Later we will clarify this statement and give formal content to these terms with the help of Petri net models. In the meantime, we will use them interchangeably.

XIX. Batching Queries and Updates

Some item-organized files can overcome their volumetric inefficiency at retrieval time by batching retrieval requests. The total volume of cross-indexing information will be transacted with for each request, but the transmission of the volume need occur only once for the set of batched requests. Edge-notched card systems have only a very limited batching capability because of the hardware characteristics of the medium. Computerized item-sequenced retrieval systems, on the other hand, have a considerably greater batching capability, determined by three principal factors: high speed memory capacity, high speed memory access time, and secondary storage to high speed storage transmission time. The discrepancy between input/output transmission speed and internal processing speed -- the second being usually much greater than the first per query per item -- creates unused processing capability which can be exploited bv batching.

We can view this phenomenon from the other side, by considering the subvolume of cross-indexing bits transacted with in an inverted file when requests are batched. The more requests we handle concurrently, the more inverted lists we will have to fetch. In the limit the subvolume becomes equal to the total volume. Hence the batching of requests reduces the volumetric differences between item-

and descriptor-organized files in retrieval. Similarly, the batching of updates reduces the volumetric differences between item- and descriptor-organized files in update processing. Even though relative volumes are only a theoretical measure of the amount of information to be transacted with, we can expect pragmatic verification for this measure. In fact, computer-based item-organized information retrieval systems generally batch requests and computer-based descriptor-organized information retrieval systems generally batch updates.

XX. An Important Asymmetry from the User's Point of View

The batching of requests in item-organized systems and the batching of updates in descriptor-organized systems have similar effects in terms of system through-put capability -- i.e., more efficient utilization of the hardware. However, they differ radically from the point of view of the system user. This is best seen in an interactive context. Under such circumstances, the user makes a request and expects an answer to his request. He may not be willing or able to generate a batch of requests before receiving any answers. In such a situation if there are multiple users of the same data base, the system may be able to batch requests across users; but this is not a certainty, and in many situations is not possible. On the other hand, when the user performs an update he does not expect an answer back in the same sense. All he demands is that subsequent requests do not result in incorrect answers. Hence, even though the user may not be willing or able to batch updates, the system can -- with internal buffering -- batch the update processing. This asymmetry is of critical importance in system design, and suggests that an interactive information retrieval system with a large data-base should almost certainly be implemented using the inverted file organization and employing internal batching of updates.

XXI. An Alternative Method of Representing Lists in Inverted File Organizations

The use of indirect and superimposed coding techniques allows us to increase the density of circled intersections in our grid model and to decrease the number of vertical lines. Hence in computer-implemented inverted file systems these techniques provide a method for increasing the density of bits with value 1 and decreasing the number of lists in the system. The computer also permits an entirely different method of increasing the density of 1-bits -- without decreasing the number of lists. This is accomplished by abandoning the exact correlation between bit position in the list and item number. Instead, each list will consist of the item numbers themselves, for exactly those items to which the descriptor applies. Since different descriptors apply to different numbers of items, the lists will vary in length. The process for intersection of lists is now computationally more complex, and in the sequel we will discuss some factors which affect the cost of this process. If the computational speed is much faster than input/output transmission time between high speed storage and secondary storage, then the increase in bit density and the consequent reduction in total bit volume to be negotiated with may easily outweigh the additional computational complexity of the intersection process.

¶ Using the same notation as in Section XI, the space in bits to represent the cross-indexing information is given by

$$V_1 = I \times D$$

where I is the number of items in the system and D is the number of descriptors (lists) in the system. If we represent each list by the method suggested above, the space in bits is given by

$$V_2 = \bar{d}_i \times I \times \log_2(I)$$

where $\bar{d}_i$ is the average number of descriptors that apply to an item and $\log_2(I)$ is the number of bits necessary to represent an item number. Hence

$$V_1/V_2 = D/\left(\bar{d}_i \times \log_2(I)\right) \quad .$$

In other words, the alternative method uses less space when the number of descriptors in the data base is greater than the product of the average number of descriptors per item and the number of bits to represent an item number. This is true of a wide spectrum of information retrieval situations. The fundamental statistical assumption that makes indirect or superimposed coding useful -- i.e., the

notion that, while there may be many thousands of different descriptors in a data base, only a very small number of descriptors will apply to any single item in the data base -- also guarantees that D is greater than $\bar{d}_i \times \log_2(I)$.

XXII. An Analogous Alternative for Item-Sequenced File Organizations

There is, of course, an analogous technique for increasing the density of 1-bits in item key words. The exact correlation between bit position and descriptor is abandoned. Instead, each item key consists of the descriptor numbers themselves, for exactly those descriptors which apply to the item represented by the item key. The item keys will now vary in length. The process for testing inclusion of the query descriptors in the item key is now computationally more complex, but once again, if there is a discrepancy between computational speed and transmission speed, the reduction in bit volume to be negotiated with may outweigh the additional computational complexity.

The space in bits required by this method is

$$V_3 = \bar{d}_i \times I \times \log_2(D) \quad .$$

Hence

$$V_1/V_3 = D/\left(\bar{d}_i \times \log_2(D)\right) \quad .$$

This technique is likely to save space in all systems that satisfy the fundamental statistical assumption

mentioned above. In fact, for a wide spectrum of systems the number of descriptors is considerably smaller than the number of items, and comparison of the formulae for V_2 and V_3 shows that in such cases more space will be saved in the item-sequenced organization than in the inverted file organization.

XXIII. A Comparison of Three Organizations for Indexing

We have now come far enough in our study to examine three commonly used file structures for indexing -- the item-sequenced file, the inverted file, and the list-structured file -- with realistic assumptions about hardware characteristics, details of representation, and usage statistics.

In an item-sequenced file each document is represented by an entry which consists of a variable number of descriptors. A query is performed by reading the whole file, determining for each entry whether the query is satisfied or not, and, if satisfied, adding the document number to a list of 'hits'. An update is performed by adding an entry to the file.

In an inverted file each descriptor is represented by a list of document numbers (for those documents to which the descriptor applies). A query is performed by reading the appropriate list for each descriptor, intersecting the lists, and ending up with a final list of 'hits'. An update is performed by adding a document number to the lists for those descriptors which apply to the document.

In a list-structured file each document is represented by an entry which consists of a document name plus a variable number of descriptor-pointer pairs. The pointer associated with a descriptor points to the next entry to which that descriptor is applicable. We assume we have identified which descriptor in the query applies to the smallest number of documents. A query is performed by reading in the first entry to which the identified descriptor applies; for the entry determining if the whole query is satisfied or not; if satisfied, adding the document number to a list of 'hits'; and in any case reading in the next entry to which the identified descriptor applies, as indicated by the pointer associated with the descriptor in the current entry. This process continues until a null pointer is encountered. An update is performed by adding an entry to the file and linking each descriptor-pointer in the entry to its appropriate descriptor-pointer chain.

Basic Formulae for the Three Design Types: Average Volume dealt with in a Transaction

(1) Item-Sequenced File:

$$V_{rI} = I\bar{d}_i \log_2 D$$

$$V_{uI} = \bar{d}_i \log_2 D$$

$$V_I = pV_{rI} + (1-p)V_{uI} = (I\bar{d}_i \log_2 D)(p(1-1/I) + 1/I)$$

(Compare this formula with Section XI, formula (g) and Section XVII, formula (b).)

(2) Inverted File:

$$V_{rD} = \bar{d}_r \left[\sum_{j=1}^{D} F(j)P(j) \right] \log_2 I$$

$$V_{uD} = \bar{d}_i \left[\sum_{j=1}^{D} F(j)P(j) \right] \log_2 I$$

$$V_D = pV_{rD} + (1-p)V_{uD} = \left[\sum_{j=1}^{D} F(j)P(j) \right] \log_2 I \left(p(\bar{d}_r - \bar{d}_i) + \bar{d}_i\right)$$

where $F(j)$ is the number of items to which the j^{th} descriptor applies and $P(j)$ is the probability distribution function of the space of all descriptor occurrences (i.e., if we observe the requests over a period of time and create a string of all requests, then $P(j)$ is the probability that at any point on that string the j^{th} descriptor occurs). (Compare this formula for V_D with Section XI, formula (f) and Section XVII, formula (a).)

If $F(j)$ and $P(j)$ are both uniform distributions, then:

$$F(j) = \bar{d}_i I/D$$

$$P(j) = 1/D$$

and

$$V_D = \left(\frac{I\bar{d}_i}{D}\right)\log_2 I \left(p(\bar{d}_r - \bar{d}_i) + \bar{d}_i\right)$$

(3) List-Structured File:

$$V_{rL} = \left(\bar{d}_i(\log_2 D + \log_2 I) + \log_2 I\right) \sum_{j=1}^{D} F(j)P_{\bar{d}_r}(j)$$

$$V_{uL} = (\bar{d}_i + 1)\left(\bar{d}_i(\log_2 D + \log_2 I) + \log_2 I\right)$$

$$V_L = pV_{rL} + (1-p)V_{uL}$$

$$= \left(\bar{d}_i(\log_2 D + \log_2 I) + \log_2 I\right)\left(p\left[\sum_{j=1}^{D} F(j)P_{\bar{d}_r}(j)\right] + (1-p)(\bar{d}_i + 1)\right)$$

where $P_n(j)$ is the probability that, when the number of descriptors in a request is n, the j^{th} description will (i) be in a request and (ii) be the lowest indexed descriptor

in the request. $P_n(j)$ is defined inductively as follows:

$$P_1(j) = P(j)$$

$$P_n(j) = \frac{P(j) + \sum_{k=n-2}^{j-1} P_{n-1}(k)}{\sum_{\ell=n}^{D} \left[P(\ell) + \sum_{k=n-2}^{\ell-1} P_{n-1}(k) \right]}$$

Note that the descriptors are indexed in descending order of list length. Because of this $P_{n-1}(k) = 0$ for $1 \leq k < n-2$.

In an item-sequenced file, since the whole file is always read when performing a retrieval, we can assume that the only seek required (in pseudo-random access secondary store, for instance) is the location of the first entry, and that the file can be read serially and hence at transmission speed. (This may require double buffering, and/or overlapped computation and I/O, and/or a fast enough computation rate to permit continuous I/O transmission, and/or very short start-stop time on the I/O gear, etc.). In other words, we assume that the time required to process the query is simply a function of one seek, serial transmission rate, and file length. We assume the time required to process an update is determined by one seek, serial transmission rate, and entry length.

In an inverted file, the set of lists to be read will in general vary from query to query. We assume that each list will require a minimum of 1 seek, followed by serial transmission of the entries on the list. The time required to process the query will be a function of the seek time, the number of descriptors in the query (i.e., the number of lists to be read), the serial transmission rate, and the individual list length. We assume the time required to process an update is determined by the number of descriptors in an item, the seek time, serial transmission rate, and individual list length.

In a list-structured file each entry read will require a seek, followed by serial transmission of the entry. The time required to perform a query will be a function of the seek time, the number of items for the most specific descriptor in the query, the serial transmission rate, and the individual entry length. We assume the time required to process an update is determined by the size of an entry, the number of descriptor-pointer pairs, seek time, and serial transmission time.

<u>Average Transaction Time for the Three Design Types</u>

$$T_I = r_a + r_t V_I$$

$$= r_a + r_t (I\bar{d}_i \log_2 D)\Big(p(1-1/I) + 1/I\Big)$$

$$T_D = r_a\left(p(\bar{d}_r-\bar{d}_i) + \bar{d}_i\right) + r_t V_D$$

$$=\left(p(\bar{d}_r-\bar{d}_i) + \bar{d}_i\right)\left(r_a + r_t\left[\sum_{j=1}^{D} F(j)P(j)\right]\log_2 I\right)$$

$$T_L = r_a\left(p\left[\sum_{j=1}^{D} F(j)P_{\bar{d}_r}(j)\right] + (1-p)(\bar{d}_i + 1)\right) + r_t V_L$$

$$= \left[p\left[\sum_{j=1}^{D} F(j)P_{\bar{d}_r}(j)\right] + (1-p)(\bar{d}_i + 1)\right]$$

$$\left[r_a + r_t\left[\bar{d}_i(\log_2 D + \log_2 I) + \log_2 I\right]\right]$$

where T_I is the average time of transaction in an item-sequenced file

T_D is the average time of transaction in a descriptor-organized file

T_L is the average time of transaction in a list-structured file

r_a is the average seek time

r_t is the serial transmission rate

XXIV. A New Method for Performing List Intersections

Our discussion thus far has concerned itself primarily with questions about the volume of information to be located and transacted with. There has been no consideration of internal computation beyond the assumption that computational speeds are always sufficient to permit continuous I/O operation. In inverted file systems, however, computation of the intersection of lists may prove sufficiently expensive to invalidate this assumption. E. Wong derives formulae for estimating the average number of comparisons necessary to calculate the intersection of two lists.[1] We repeat his derivation here:

First, let A and B be unordered lists. Then, assuming uniform distribution for the location of a_i in list B, it takes an average of $n_B/2$ comparisons to find a_i, if a_i is in B. If a_i is not in B, it takes n_B comparisons to ascertain this fact. The average number of comparisons is then

$$T_1 = n_{AB}\frac{n_B}{2} + (n_A - n_{AB})n_B$$

$$= (n_A - \frac{n_{AB}}{2})n_B$$

---

[1] E Wong, Time Estimation in Boolean Index Searching (December 1961, in High Speed Document Perusal, AD 285 255).

where n_{AB} is the number of elements in the intersection of A and B.

When A and B are ordered lists, a logarithmic search procedure can be adopted (i.e., successively dividing B into equi-probable subsets). Again assuming a uniform distribution, the number of comparisons needed for each a_i is approximately $\log_2 n_B$ if $a_i \varepsilon B$ and $\log_2 n_B + 1$ if $a_i \not\varepsilon B$. The average number of comparisons is then

$$T_2 = n_{AB}\log_2 n_B + (n_A - n_{AB})(\log_2 n_B + 1)$$

$$= n_A \log_2 n_B + (n_A - n_{AB})$$

In order to calculate an intersection it is necessary to determine of an element, $\bar{a}$, of list A whether or not $\bar{a}$ is a member of B. This is a familiar problem and immediately suggests the use of hash (or scatter storage) tables. We propose the following procedure:

Suppose $n_B \leq n_A$. Create a hash table containing all members of B. For each element, $\bar{a}$, of A use the hashing procedure to decide whether $\bar{a}$ is a member of B. $A \cap B$ contains all elements of A for which this decision is yes.

¶ We observe that the average cost has three components.

T_{31} = cost of creating hash table for B

T_{32} = cost of deciding for elements of A which are in B

T_{33} = cost of deciding for elements of A which are not in B

To obtain an explicit formula for this cost it is necessary to choose a particular hash technique. For this application the following technique is adequate:[2]

(1) Generate a hash address from an entry by squaring the entry and choosing some bits from the center of the square.

(2) Resolve collisions by random probing.

We can now exhibit explicit formulae of cost:

$$T_{31} = n_B(c_h + (E - 1)c_r)$$

where

[2] See Morris, R., "Scatter Storage Techniques", Communications of the ACM, January 1968.

c_h = cost of generating a hash address

c_r = cost of random probe

E = average number of probes necessary to hash an entry of B

$= -\left(\frac{1}{\alpha}\right)\log(1 - \alpha)$

α = load factor of the hash table

$= n_B/N$

N = size of hash table

(See Morris for the derivation of E and of A below.)

If an element of A is also in B then the cost of the decision is the same as the average cost of hashing an element of B. Hence

$$T_{32} = n_{AB}\left(c_h + (E - 1)c_r\right)$$

If an element of A is not in B then the cost of the decision is the same as the cost of adding a new element to the hash table containing all of B. Hence

$$T_{33} = (n_A - n_{AB})\left(c_h + (A - 1)c_r\right)$$

where $A = \dfrac{1}{1 - \alpha}$

now $T_3 = T_{31} + T_{32} + T_{33}$

$$= n_B\left(c_h + (E-1)c_r\right) + n_{AB}\left(c_h + (E-1)c_r\right)$$

$$+ \left(n_A - n_{AB}\right)\left(c_h + (A-1)c_r\right)$$

$$= (n_A+n_B)(c_h-c_r) + c_r\left(E(n_B+n_{AB}) + A(n_A-n_{AB})\right)$$

For example, suppose we allocate space for a hash table one third larger than list B. Then

$$\alpha = \frac{n_B}{N} = .75$$
$$E = 1.83$$
$$A = 4$$

Let us assume that

$c_h \cong 4$ comparisons

$c_r \cong 2$ comparisons

Then

$$T_3 \cong 10n_A + 5.66n_B - 4.34n_{AB} \text{ comparisons}$$

Let us now compare T_3 with T_1 and T_2. For purposes of comparison let $n_A =$ [illegible]

	$n_{AB}=0$	$n_{AB}=\frac{1}{2}n_A$	$n_{AB}=n_A$
T_1	$n_A \times n_A$	$(\frac{3}{4}n_A) \times n_A$	$(\frac{1}{2}n_A) \times n_A$
T_2	$\log_2(2n_A) \times n_A$	$\log_2(\sqrt{2}n_A) \times n_A$	$\log_2(n_A) \times n_A$
T_3	$15.66 \times n_A$	13.[illegible] $\times n_A$	$11.32 \times n_A$

From this we can observe that if $n_A > 23$, then $T_3 > T_1$ irrespective of the number of elements in the intersection. $T_1/T_3 = \frac{n_A}{22.64}$ under the most favorable circumstances for T_1 in the comparison. As n_A increases, T_1 becomes proportionately larger than T_3. Similarly, if $n_A < 2^{11.32}$, then $T_2 > T_3$ irrespective of the number of elements in the intersection.

For a wide spectrum of information retrieval systems, list length will be less than 2^{11} for most lists; hence if maintaining the inverted lists in <u>sorted</u> order incurred no additional cost, method 2 would be preferable. In systems which permit deletes, it may be extremely expensive to do this, and it would be necessary to judge this expense against the possibility of using method 3 (which improves relative to method 2 as the data base grows). Method 3 is almost always preferable to method 1, and system design evaluations based on method 1 are grossly

unfair to inverted files.[3] Note that it will not do to sort the lists immediately prior to intersection. This added cost would make T_2 worse than T_3 for virtually all cases.

---

[3] M. Kochen, Preliminary Operational Analysis of a Computer-Based, On-Demand Document Retrieval System Using Coordinate Indexing.

XXV. Data Compression -- Another Encodement for Inverted Lists

In Section XXI we discussed a method for increasing the density of 1-bits in each inverted list which abandoned the exact correlation between bit position and item number. Each list consisted of the item numbers themselves, for exactly those items to which the descriptor represented by the list applied. The volumetric effects of this technique were discussed in Sections XXI and XXIII. It is possible to compress lists even further, by taking advantage of 'burst' characteristics within a list.[1]

In our previous calculations we have assumed that the number of bits needed to represent an item in an inverted list is $\log_2 I$, where I is the total number of items in the system. Instead, we can use the following encoding procedure for an inverted list:

(1) Record the first item number as a $(\log_2 I)$ bit quantity.

(2) Take the difference between adjacent item numbers in the list.

(3) If there is one or more consecutive differences

---

[1]See Computer Programming Techniques for Intelligence Analyst Application. AD 608 727, October 1964.

of 1 (i.e., two or more consecutive item numbers), record a 6-bit code (SC_1), followed by the number (+1) of consecutive numbers (less than 61).

(4) If step 3 does not apply but the difference is less than 61, record the difference as a 6-bit quantity.

(5) If the difference is greater than 61 but less than or equal to 4095, record a 6-bit code (SC_2) followed by the difference as a 12-bit quantity.

(6) If the difference is greater than 4095, record a 6-bit code (SC_3) followed by the difference as a ($\log_2 I$) bit quantity.

The authors of Computer Programming Techniques for Intelligence Analyst Application show a slightly more than twofold reduction in their example as a result of this technique.

This technique could be applied selectively. In particular, certain inverted lists will tend to grow large (some authors suggest a Zipf distribution of inverted list lengths.) The compression would tend to be most effective for such lists in which the density of applicable items is highest. Furthermore, for a wide spectrum of information retrieval systems in which item numbers are assigned sequentially on input, there is reason to expect

bursts to occur as a result of additions to the system of groups of items related to the same subject or subjects. In any case, by reserving one bit at the beginning of each inverted list, the system can adaptively decide for each list whether the compression technique should be employed. The bit informs the system -- at retrieval or update time -- how to interpret the list. This _may_ involve more computation time and this must be weighed against the reduction in the number of bits transmitted.

XXVI. A Grammar for Defining Graph Representations of File Structures

In this section we present a formal apparatus for defining models of file structures. We do not pretend to have mathematical techniques which permit a formal analysis of models defined in this way. At this point in time, the formal apparatus serves only as a definitional vehicle -- questions of cost and efficiency of utilization under varying usage statistics and hardware milieus must be answered by performing a mathematical analysis based on formulae such as those derived in earlier sections of this report. Nevertheless, a formal apparatus for <u>defining</u> models of file structures is of value, because such definitions can provide significant insights and facilitate the comparison of different structures meant to accomplish the same task.

The state of a system is formalized as a finite undirected graph with labeled nodes; every node has a label, called the <u>node type</u>, and several nodes may have the same label. Such a graph, when intended to represent a system state, is called a <u>configuration</u>. The nodes are interpreted as system parts; an arc is interpreted as a relation between two parts. The labels correspond to classes of parts which have identical possible contexts. Thus the nodes corresponding to the cells of a memory might have identical

labels, for each cell has the same contextual possibilities: each stands in relation to exactly one value and exactly one of the cells stands in relation to a memory exchange register. It will be observed that "part" means logical part, not physical part; a part is anything which can participate in an observable relation. Parts may be things, values, states, etc.

The appearance of an arc between two nodes asserts the possibility of conditioning some occurrence, internal or environmental, upon a relation between the two corresponding parts.

The formal apparatus of η-grammar is built around the <u>site-spec</u>, which -- like the configuration -- is a finite undirected graph with labeled nodes.

A η-grammar defines a class (perhaps infinite) of configurations. Members of this class are said to satisfy the grammar or be <u>grammatical</u>. The set of configurations which satisfy a grammar corresponds to the class of possible system states for the family of discrete information systems described by the grammar.

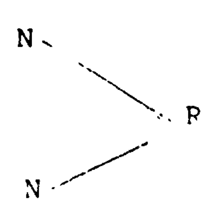
A η-grammar sets forth the local laws which constrain relation among parts; for example:

- Every row, column pair has exactly one value holder.
- Every bit position holds either zero or one, exclusively.
- Every integer has a unique successor or is marked with a "last" marker, exclusively.
- Every control counter holds exactly one address.
- Every binary tree element has at most two successors.
- Every value may be held by an arbitrary number of value holders.

Formally, a 𝔇-grammar is a finite list of grammar rules. There are two types of grammar rules:

Type A rule: a site-spec which properly contains a single circled site-spec.

Type B rule: a site-spec containing no circled site-spec.



We call the circled site-spec in type A rule the subject of the rule. A rule of which s is a subject will be referred to as an s-rule.

Site-Spec Satisfaction

We say that a site-spec S is satisfied in a configuration C , or S has a satisfaction in C , if there exists a 1-1 map M from the nodes of S into the nodes of C such that the type of every S-node is the same as that of its image under M , and for every arc between nodes of S there is an arc between the corresponding image points in C . The map M is called a satisfaction map or satisfaction of S in C . Any arbitrary collection of nodes in a configuration is called a place, and the image of S under M is in particular called a place of satisfaction.

A place p in a configuration obeys a rule r if (1) it satisfies the subject of the rule, and (2) is contained in a place p' which satisfies the rule, with the two maps agreeing on the subject; p' is then referred to as a place where p obeys r .

Grammaticality

A configuration is said to be grammatical if and only if:

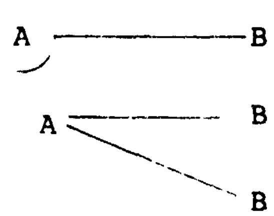
(1) If a place p satisfies a subject s in the grammar, then p obeys at least one s-rule in the grammar.

(2) For every arc (p,q) in the configuration, there must exist a rule r satisfied in a place including p and q , such that the inverse images p' and q' under the satisfaction map are connected by an arc in r .

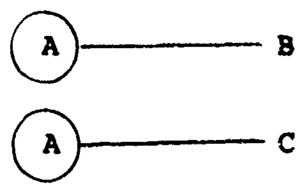
(3) There are <u>no</u> satisfactions of type B rules in the configuration.

The way in which the definition works might best be clarified by example.

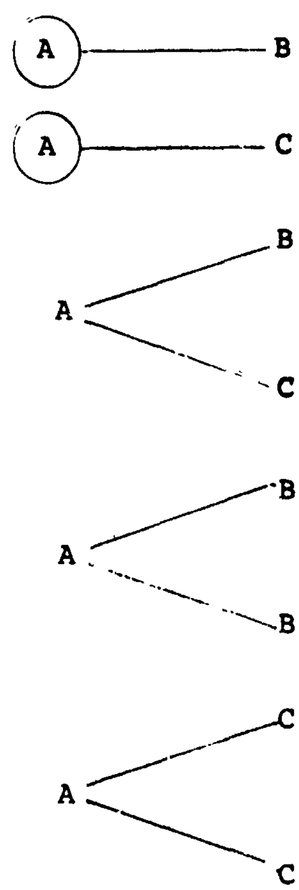
In the absence of other rules with the same subject, this rule asserts that every A has at least one B.



Every A has exactly one B.



Every A has at least one B or at least one C or at least one B and one C.



Every A has exactly one B or exactly one C, but not both.

<u>A Simple File Structure</u>

Consider the following (informally characterized) file structure:

1. A file consists of named records and

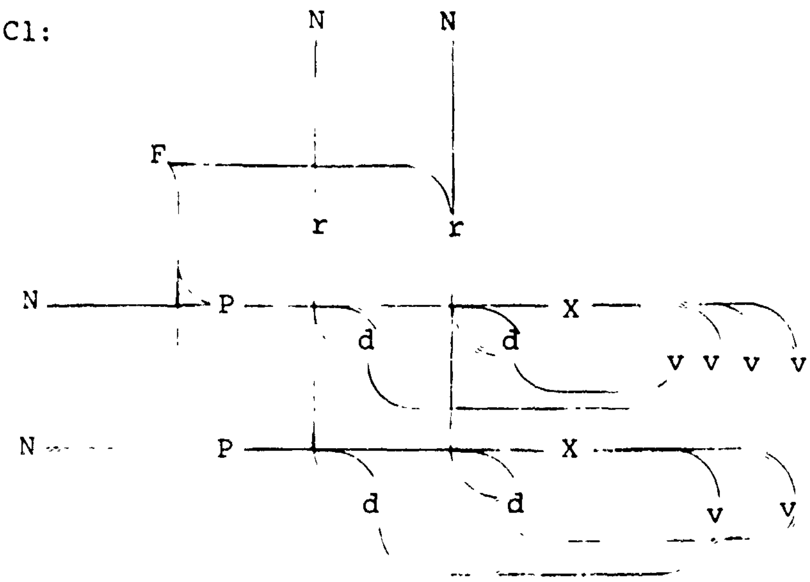
named properties.

2. No two records (or properties) of the same file may have the same name.
3. Each record has one data position for each property of the file.
4. Each property specifies a domain of possible values.
5. Each data position always contains some one value belonging to the value domain specified by the property corresponding to that data position.

C1* is a configuration which represents an instance of such a file structure.

* To facilitate the illustration of configurations a convention has been adopted to reduce the number of arcs in the drawing. According to this convention, two nodes are connected (they "associate") if it is possible to reach one directly from the other without turning sharp corners. Hence in C1 all records are connected to the file, but not to each other.

C1:



Mnemonic Aid

d ∿ data position

F ∿ file

N ∿ name

P ∿ property

r ∿ record

v ∿ value

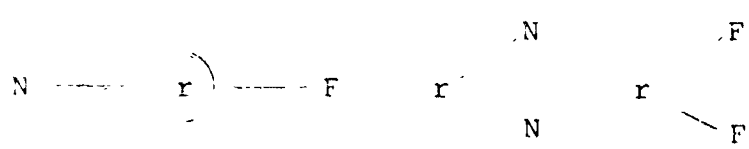
X ∿ value domain

In this instance the file consists of two records and two properties. One property specifies a value domain consisting of four values; the other specifies a value domain of two values.

The following grammar suffices for defining the class of configuration informally characterized above:

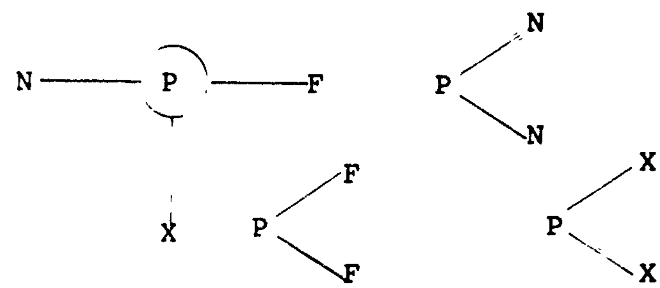
1: 

A file has at least one record and property.

2: 

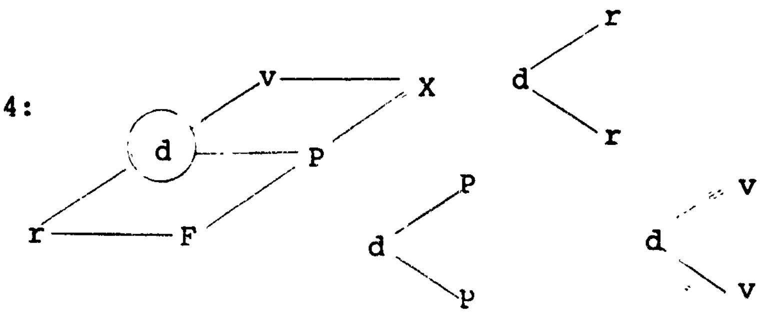
A record belongs to exactly one file and has exactly one name.

3:



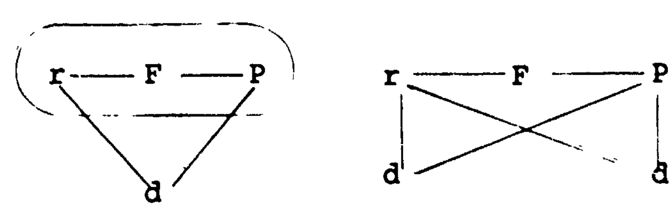
A property belongs to exactly one file, has exactly one name, and specifies exactly one value domain.

4:



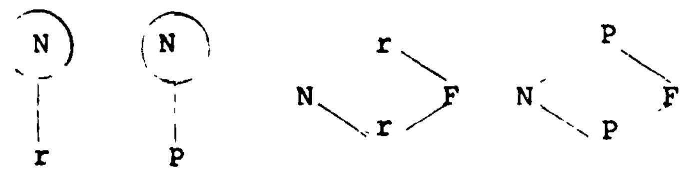
A data position is coordinated by exactly one (record, property) pair of a file and contains exactly one value from the value domain specified by the property.

5:



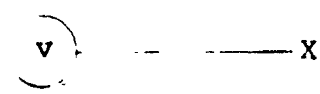
A (record, property) pair of a file coordinates exactly one data position.

6:



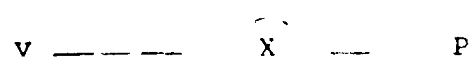
A name associates with at least one record or at least one property or both. No two records (properties) within the same file may have the same name.

7:



A value associates with at least one value domain.

8:



A value domain associates with at least one value and at least one property.

It is interesting to consider briefly some interpretations of the class of configurations defined by the above grammar.

Suppose that a list of event types is defined such that the only change possible in a configuration is the

reassociation of data positions with values. In other words, given a starting configuration, the number of records and properties in a file would remain constant, the relations between properties, value domains, and values would remain fixed, but the particular value associated with each data-position could change.

A conventional fixed length, fixed format table behaves in such a manner. The lines in the table correspond to records. The fields in the table correspond to properties and the fields specify a value domain by virtue of the number of bits allotted per field. The only variable aspect of such a table is the set of values contained in the field positions on each line.

An Interpretation of Configuration C1 as an instance of a Fixed-Length Fixed-Format table:

	Gene	Tom
Line 1	2	0
Line 2	0	1
	2 bits	1 bit

Gene is the name of a property which specifies a 2-bit domain.

Tom is the name of a property which specifies a 1-bit value domain.

Line 1 is the name of a record.

Line 2 is the name of a record.

Suppose now that the list of event types is extended to permit addition and deletion of records to the file.

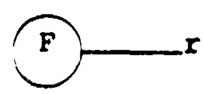
A conventional, simply-formatted tape file behaves in such a manner. Every record in the file has the same format: each record consists of a set of values, one per property as specified by the format. The only variation from record to record is the particular value set. However, the number of records in the file is permitted to vary.

Suppose the list of event types is further extended to permit addition and deletion of properties and value domains. A system with dynamically definable variable length tables would behave in such a manner. A data-base with dynamic restructuring capabilities might also be characterized in this way.

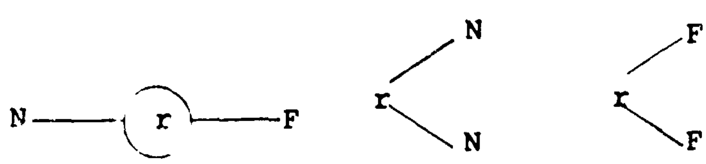
File Hierarchies

The grammar presented above provides a basis for constructing more sophisticated systems. As an example, the following grammar defines a class of configurations

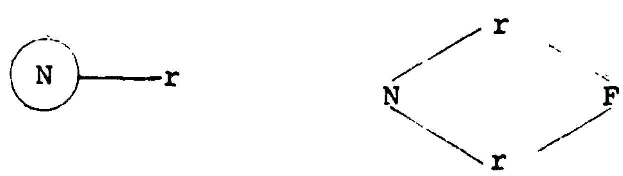
suggestive of the data-base structures obtainable in ADAM.[1]

1: 

(From rule 1 of previous grammar)

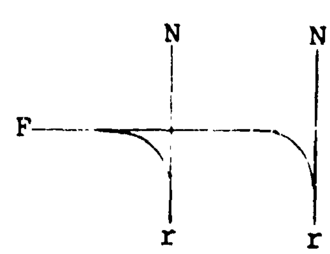
2: 

(From rule 2)

3: 

(From rule 6)

C2 is a configuration from the class defined by the above rules.

C2: 

<u>Mnemonic Aid</u>

F ∿ file

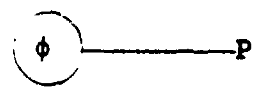
N ∿ name

r ∿ record

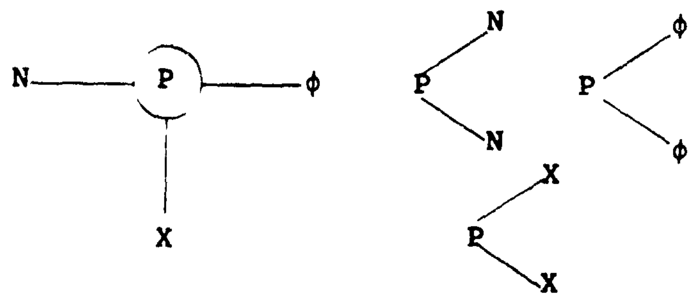
We will call this class of configurations a File Unit.

---

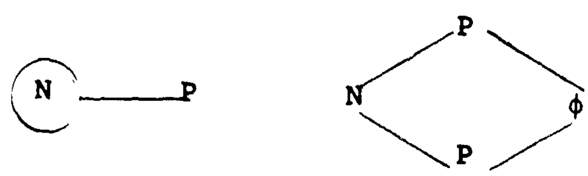
[1]<u>ADAM - A Generalized Data Management System</u>. Paper presented at SJCC 1966, by T.L. Conners, The Mitre Corporation.

4: 

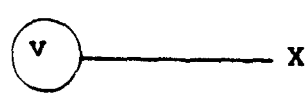
A format has at least one property.

5: 

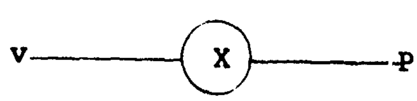
(From rule 3, previous grammar)

6: 

(From rule 6)

7: 

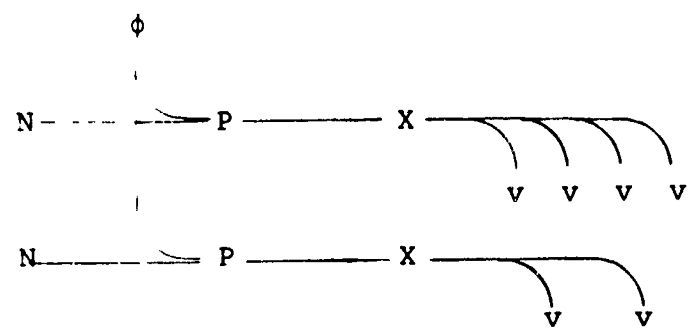
(From rule 7)

8: 

(From rule 8)

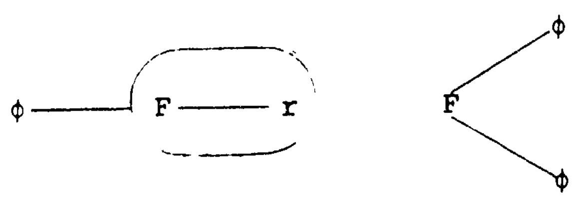
C3 is a configuration from the class defined by rules 4 through 8:

C3:



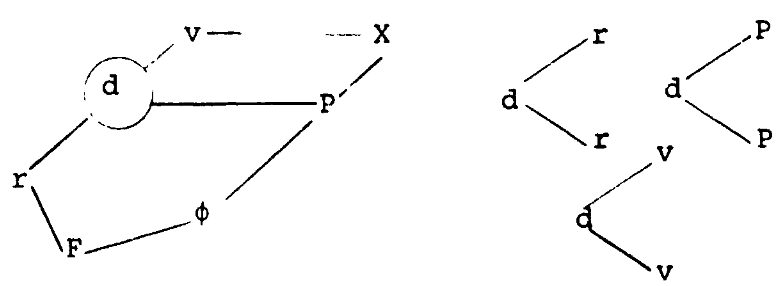
We will call this class of configurations a Format Unit.

9:



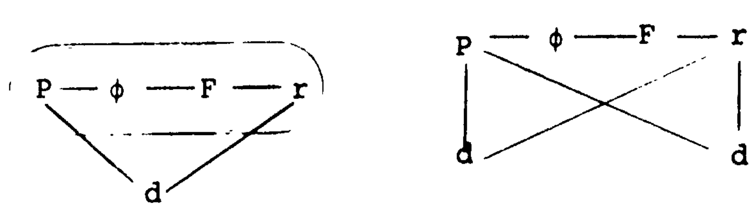
In combination with rule 1, this guarantees that every file will have exactly one format.

10:



(From rule 4, previous grammar)

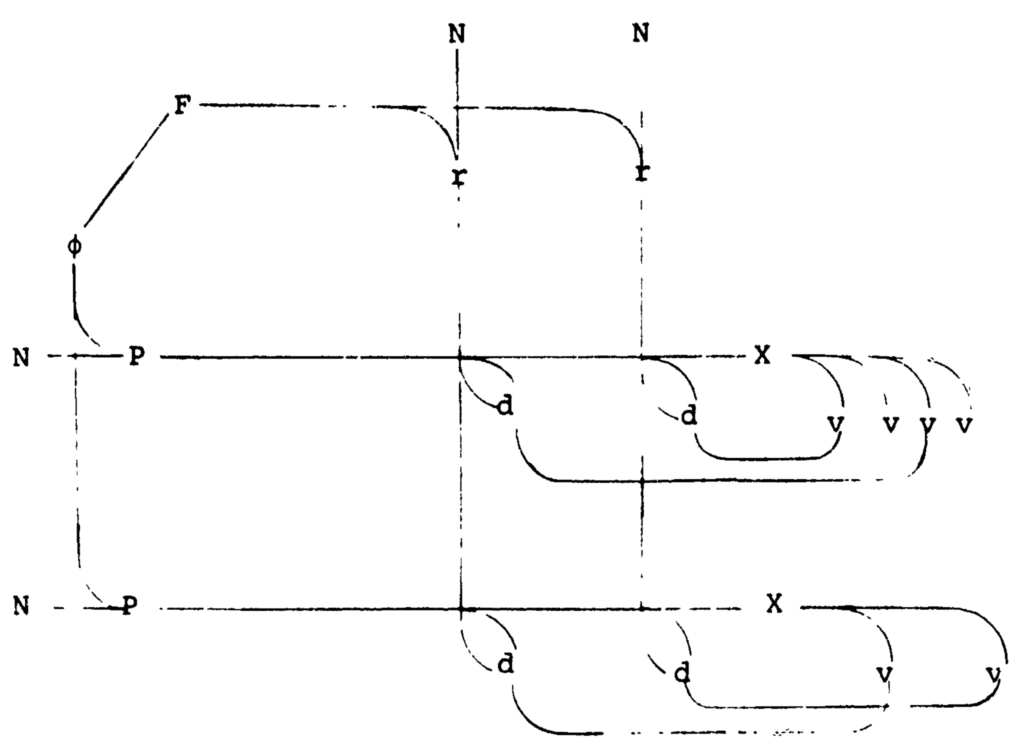
11:



(From rule 5)

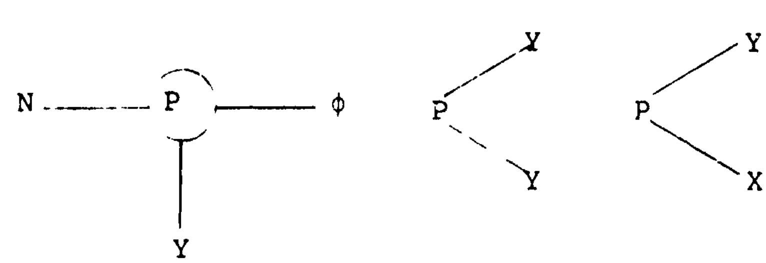
Rules 9 through 11 may be thought of as defining the way in which File Units and Format Units connect. C4 is a configuration from the class defined by rules 1 through 11. Refer back to configuration C1 for comparison.

C4:



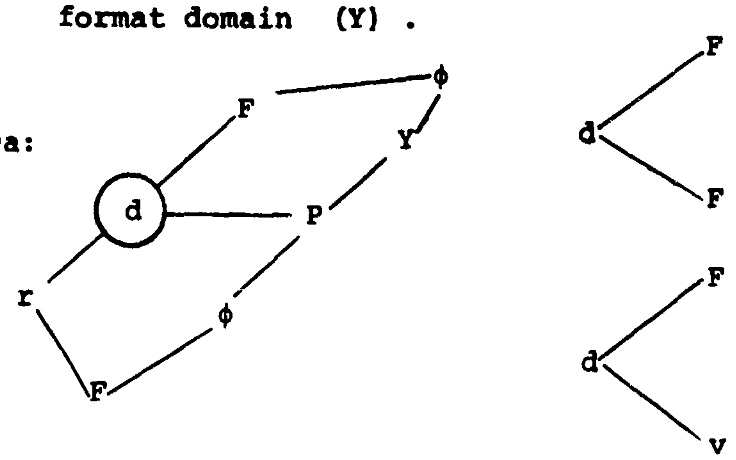
We now extend the grammar rules to permit file hierarchies.

5a:



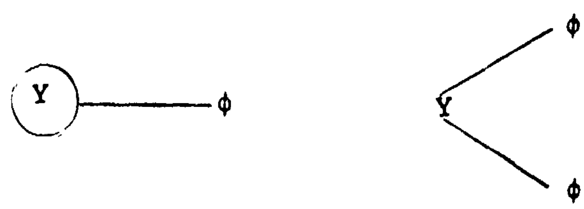
Previously, a property specified exactly one value domain. The addition of the rules in 5a permits a property to specify either exactly one value domain <u>or</u> exactly one format domain (Y) .

10a:



Previously, a data position contained exactly one value from the value domain specified by the associated property. The addition of the rules in 10a permits an alternative: a data position may specify exactly one file; the format of that file relates to the property associated with the data position, via the unique intermediary format domain Y .

12:

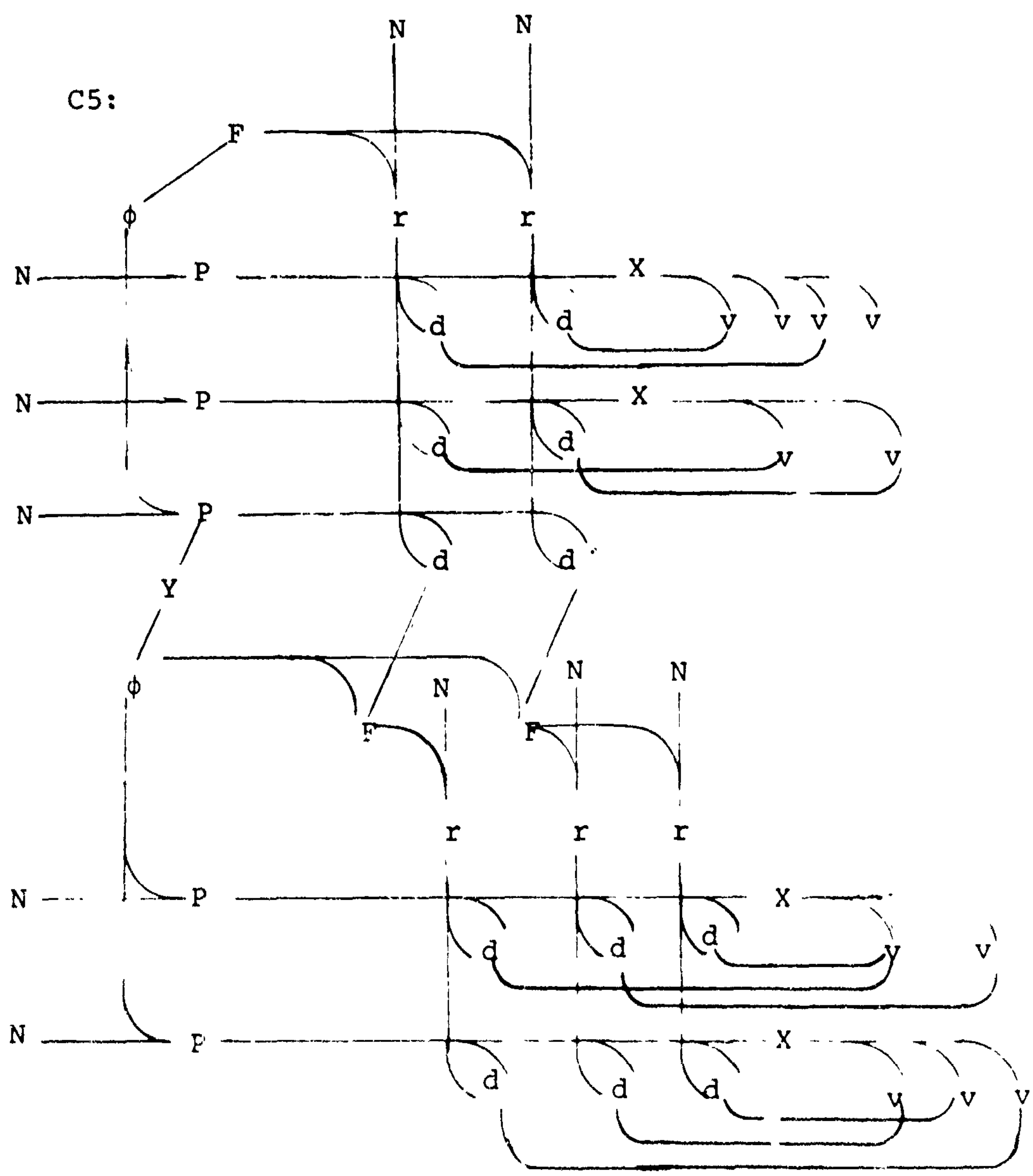


A format domain associates with a unique format.

Rules 5a and 12 may be thought of as defining the way in which Format Units connect to each other. Rule 10a extends the way of connecting File Units to Format Units and allows files to have sub-files.

Rules 1 through 12 define a class of configurations of which C4 is surely a trivial instance. A more interesting example is configuration C5.

C5:



Mnemonic Aid

d ∿ data position

F ∿ file

N ∿ name

P ∿ property

r ∿ record

v ∿ value

X ∿ value domain

Y ∿ format domain

ϕ ∿ format

Sub-files as represented here are analogous to ADAM repeating groups.[2]

In configuration C5 we have represented a file with two records. The format of that file requires that each record of the file have two data holders that contain unique values and one data holder that specifies a unique sub-file. The values must be chosen from the appropriate value domain, as specified by the properties of the format. All sub-files must have the same sub-format as specified by the single P———Y in the format.

Hence, the two sub-files exhibited are forced to associate with the appropriate sub-format, and this in turn guarantees that their records (one sub-file has a single record, the other has two records) all have two data holders containing unique values from the appropriate domains.

Once the Format Units have been chosen, the grammar allows (1) the number of records in any file to be variable, and (2) the free choice of which value (in the appropriate domain) a data holder associates with; but the grammar fixes all other aspects of the structure.

---

[2]Conners, op cit.

XXVII. A Critique of Balanced Trees

In preceding sections we have discussed cross-indexing in considerable detail, but without ever directly considering the problem of converting a descriptor into a code or list address. All of the computer-based indexing systems which we have dealt with require some sort of dictionary to perform this conversion. In this section we will examine the use of balanced trees -- the technique employed to accomplish this conversion in MULTI-LIST. As is often the case with information retrieval systems, the justification for this technique is based on a number of questionable assumptions about usage statistics and costs, and we shall want to challenge these assumptions or at least make explicit their implications. We shall then show that, even granting these assumptions, there are other, more efficient techniques. Finally, we will propose an alternative technique.

The following quote will give some idea of the relative importance of Landauer's work in the MULTI-LIST system:

> The topic of this dissertation evolved from the research on new methods of computer memory organization that potentially lend themselves to efficient information storage and retrieval. An important place in this area is held by the MULTI-LIST organization of the memory, which is an extension of the list-type associative memory conceived originally by Newell, Shaw, and Simon for the simulation of human thought processes in learning and problem solving. The tree, which is a basic building block of the MULTI-LIST system,

> constitutes the central notion of this dissertation.[1]

Let us, then, briefly outline the use of trees in MULTI-LIST. MULTI-LIST grows a tree for the set of descriptors in the system. All nodes of the tree will have some fixed number m of branches. The tree is grown (i.e., descriptors are added) in such a way that the tree is always balanced -- that is, each level is completely filled before the next level is begun. Thus every path through the tree visits either n or n-1 (the lowest level may be incomplete) nodes, where n = the number of levels in the tree. Each node consists of m (m = the number of branches at each node) catenae. Each catena consists of a key and a pointer to some node in the next level; the pointers of emerging branches point to lists. When the system must locate the list for some descriptor (i.e., in processing a query), the tree is used as follows: The descriptor is compared arithmetically with the key of the first catena of the root node; if it is less than or equal to the key, the pointer in that catena is followed to a node in the next level. If the descriptor is greater than the key in the first catena, it is compared to the key in the second catena of the node; if

---

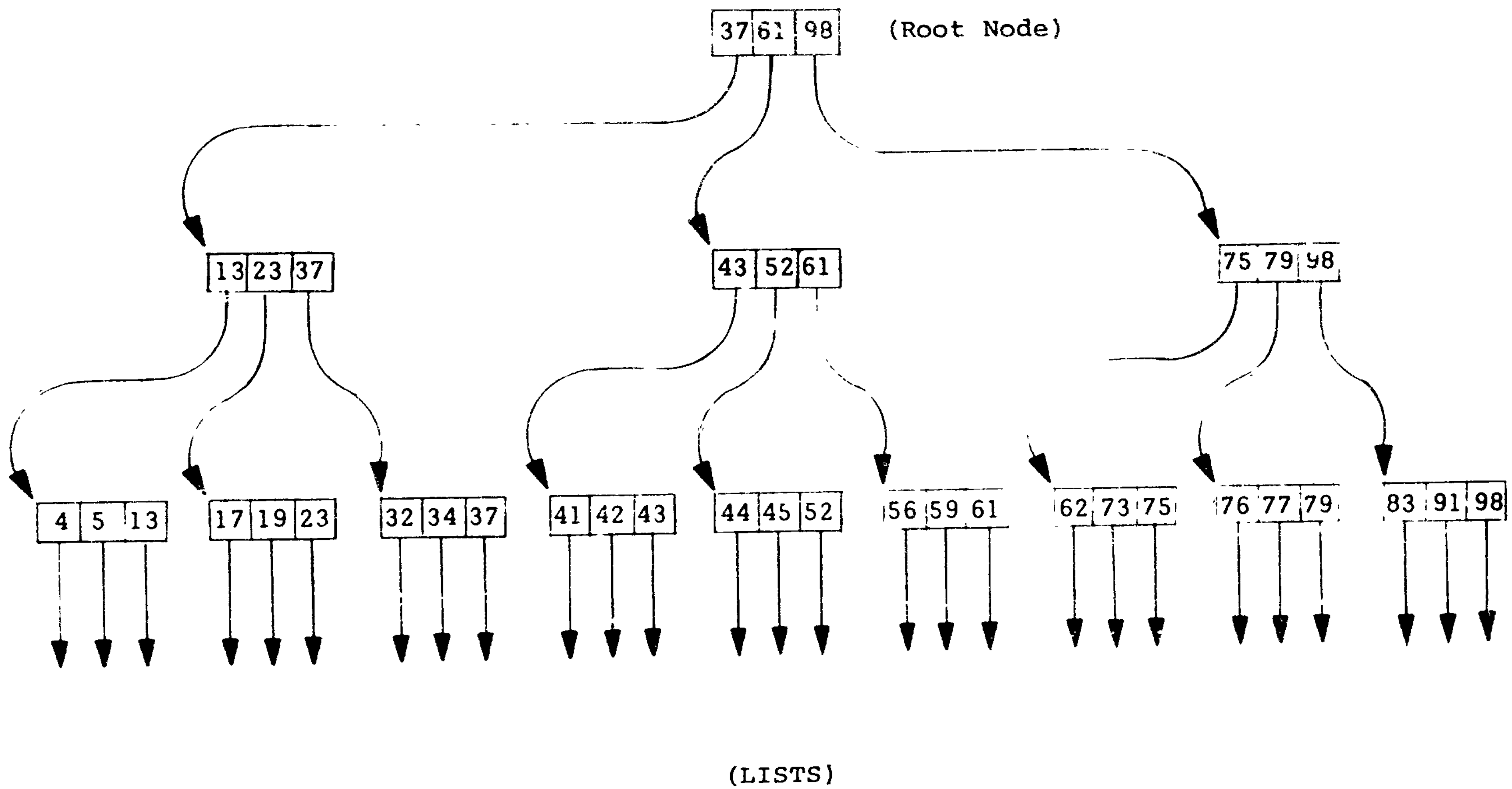
[1]Walter I. Landauer, The Tree as a Stratagem for Automatic Information Handling, Ph.D. Thesis, University of Pennsylvania, 1962. Page xii.

it is less than or equal to that key, the associated pointer is followed; if not, the descriptor is compared to the key in the next catena; and so forth. The tree is constructed in such a way that the descriptor must be less than or equal to at least the last key. The same procedure is followed at the next node encountered, and so forth. The pointer taken at the lowest level (i.e., upon emerging from the tree) will lead to the list for the descriptor. Suppose, for example, that the possible descriptor names are the integers, and that we have a system which thus far contains the following descriptors: 4, 5, 13, 17, 19, 23, 32, 34, 37, 41, 42, 43, 44, 45, 52, 56, 59, 61, 62, 73, 75, 76, 77, 79, 83, 91, 98. Figure XXVII-1 shows a tree for this example, with m (i.e., the number of branches at a node) = 3. n (= the number of levels) = 3, and F (=the number of emerging branches) = 27.

Landauer derives cost measures for such tree structures -- primarily in order to determine the optimum value for m . He begins by asserting that retrieval is the only operation which need be considered in evaluating system costs:

> The management of an information handling system involves three basic operations: filing, retrieval and deletion of an item. Whereas filing and deletion are operations that keep the file updated, and are therefore inherently "one shot" operations, retrieval will in all probability be a recurrent operation, i.e., a single item

Figure XXVII-1



> may be retrieved many times within its lifetime in the file. Consequently the preponderance of the retrieval operation with respect to speed and efficiency is obvious. Hence the conditions for optimum efficiency will be obtained from the task of information retrieval, specifically from a single search of the file.[2]

Landauer then defines the cost of a search as the product of the time per search (which in turn is defined solely by the number of comparisons made during the search) and the system cost. System cost is assumed proportional to memory size alone.

The average search time T (i.e., the average number of comparisons per tree-traversal) is given as:

$$T = n \sum_{i=1}^{m} \frac{i}{m} = \frac{\log_e F}{\log_e m} \frac{m+1}{2}$$

The cost of the tree in catena units is:

$$C = F + \frac{F}{m} + \frac{F}{m^2} + \ldots + \frac{F}{m^{n-1}} = \frac{m(F-1)}{m-1}$$

The product of C and T is the inverse of the efficiency I

$$I = C \times T = \frac{m(F-1)}{m-1} \frac{m+1}{2} \frac{\log_e F}{\log_e m}$$

---

[2]Landauer, op cit. Page 13.

" $\frac{dI}{dm}$ yields a minimum at $m = 5.25$ approximately. This value of m is the branching factor corresponding to an optimum in retrieval efficiency."[3] On the other hand, "the differentiation of the traversal-time, T with respect to m yields a minimum at $m = 3.5$ approximately. However, a plot of the corresponding curve shows that the minimum is rather broad and $m = 5.25$ that was obtained above to minimize the product of cost and time lies well within the range of the broad traversal-time minimum."[4]

We would like to raise three fundamental objections to Landauer's arguments. Let us first deal with his assertion that only retrieval is significant in evaluating system efficiency. To begin with, the characterization of filing and deletion as "inherently 'one-shot' operations" cannot possibly be a universal property of all information-handling systems. On-line command and control or intelligence systems require continuous "real-time"updating facilities, and the filing and deleting of information which is never actually "retrieved" can be a common phenomenon. Promoters of MULTI-LIST have in fact argued for its use over other systems in such contexts just because of the "relative

---

[3] _Ibid_, page 15.

[4] _Ibid_, page 16.

ease of update" in MULTI-LIST. By leaving out the cost of filing and deletion in what follows, Landauer, in effect, restricts the validity of his work to situations in which (presumably after some initial phase of system creation) there are no updates. (The reader is referred to the formulae in Section XXVII, in which the role of the retrieval/update ratio is explicitly expressed.) Suppose we accept this assumption. If filing and deletion are not cost factors, we may structure the information in any way we choose, without having to account for the cost of the structuring procedure. We might then, for example, consider the following structure: a single sorted list of associative catenae, ordered by key magnitude, just as Landauer suggests, but searched by binary search technique instead of serial comparison. In this case the space required for associative catenae is clearly a minimum (equal to F, the number of lists that emerge from the tree). This arrangement can be viewed as identical to Landauer's prescription, but with m set equal to F so that we have a one-level tree. The number of comparisons in a traversal using binary search technique will of course be $\log_2 F$, rounded up which is a significant improvement over Landauer's $(F+1)/2$. In fact -- if we correct Landauer's formula for the number of comparisons per search (see below) -- $\log_2 F$ will correspond roughly to the cost of a MULTI-LIST search through a tree with

m = 2 (i.e., a binary tree), which is a minimum for the number of comparisons. Hence the single sorted list of associative catenae, ordered by key magnitude and searched by utilizing binary search technique, requires no more space than the most space-conservative balanced tree and no more comparison time than the most comparison-conservative balanced tree. In short, if we accept Landauer's assumptions and cost-criteria, it is superior to the balanced tree scheme regardless of the value of m -- whether it be 2, 5.25, 3.5, or any other number.

It may be expensive to generate or maintain a fully sorted list of this size, but Landauer's formulation of the problem precludes considering generation or maintenance as cost factors. Again, there may be difficulties associated with the fact that a single sorted list cannot fit into main memory, but Landauer assumes that the memory is a one-level, truly random access device. If there is a set of assumptions about usage statistics and hardware which justify Landauer's conclusions, they are certainly not the assumptions which he makes. In fact there may be no assumptions which lead to his conclusions.

Our second fundamental objection is to Landauer's definition of the cost of a search as the product of the number of comparisons and the "system cost", where the latter is

proportional to the amount of memory required. Clearly no computer installation operating in batched sequential mode would ever charge rates based on such a formulation. One cannot in general assume that there will be a reduction in real cost when less than the available memory is used. Even a time-shared facility could not charge rates on such a basis. For example, in any real computing milieu, main memory is limited; if it is overflowed, input/output delays may cause enormous increases in time when secondary storage is utilized. Landauer's formulation, of course, does not evaluate such possibilities. In general, the relative cost of memory space and computational time will vary enormously from one computing milieu to another, so that Landauer's product -- although mathematically very convenient -- will not be very useful as a measure of efficiency.

Before leaving MULTI-LIST we must raise one further objection to Landauer's formulations. Based on the assumption that the keys of a node have equal chances of being selected, Landauer asserts that the average number of comparisons at a node will be $\frac{m+1}{2}$. However, there is no need to compare a descriptor against the m^{th} catena of a node (except, possibly, at the lowest tree level): if the first $(m-1)$ comparisons fail, merely follow the m^{th} pointer. In other words, the average

number of comparisons is $(m+1)/2 - 1/m$ (except, possibly, for lowest level nodes), and the computation time involved must lie between $(m+1)/2$ and $(m+1)/2 - 1/m$. The corrected probability estimate yields (for trees with large F, at least) a comparison time minimum at $m = 2$ i.e., a binary tree -- and not at $m = 3.5$. This adjustment also has repercussions for the evaluation of the minimum for Landauer's product-formulation of efficiency, of course.

This critique would be incomplete if we did not suggest some alternative to the balanced tree as a decoding technique for descriptors. We have already suggested a formulation involving binary search techniques applied to a single sorted list, which, given Landauer's assumptions, is superior to his solution. However, the critical factors of generation and update, ignored by Landauer, are known to be expensive functions when working with a single sorted list. The following technique is superior both for Landauer's rather narrow assumptions and for more general assumptions.

Reference keys (i.e., descriptors) are decoded by using hash techniques. Costs here depend upon calculation time for the hash function, search time in the hash table, and storage space required by the table. Extra space is

required in the table in order to maintain low search times, but it has been shown that the use of a good hash function (such as radix conversion) permits 80% utilization of the hash table, with searches requiring on the average less than two comparisons in the hash table. Entries in the table will contain the key and a pointer to the appropriate list. Hence the table will require only slightly more space than the fully sorted list (Landauer's $m = F$ case). Time will be approximately two comparisons + hash function evaluation. The larger F is, the more favorably this technique compares with binary search and balanced tree search. There is no need for sorting. Hash table expansion and contraction are the only updating functions that have a cost significantly greater than decoding. (Addition or deletion of a single key involves only slightly more cost than decoding a key. Repeated addition or deletion, analogous to generation and update, involve table expansion and contraction.) Table expansion is handled by recognizing when the hash table is 80% full, then requesting that the hash function increase the modulus it is using (i.e., increase hash table size), rehashing the current entries in the table, and continuing from there. Contraction is handled by the same procedure, except that the modulus is decreased when the table is sparse. The whole hash table need not fit in core: high order bits of the evaluated hash unction determine the

appropriate segment of the hash table. A simple mechanism prevents oscillation between two adjacent hash table sizes. In the next section we will discuss this technique in more detail.

XXVIII. Hashing and Secondary Storage

This section is concerned with the use of scatter storage, or hash techniques to retrieve information from large data bases. We will draw a distinction between scatter index tables, where each entry contains a pointer to the desired item, and scatter storage tables, where each entry is the desired item. We are particularly interested in the case where both the data base and a scatter index table giving access to the data base are so large that they cannot be contained in core. This situation commonly occurs in the environment of a time sharing system on a paged machine with a disc or drum secondary store. In such an environment the entire data base would reside in secondary; a scatter index table would primarily reside in secondary but it would be possible, in general, to lock a few pages in core. We will be concerned with two formats for the data base: fixed length and variable, unbounded length. Since disc access times and transfer times are typically 4 to 5 orders of magnitude greater than memory cycle times we are primarily concerned with minimizing the number of disc accesses so as to both increase throughput rate of the information retrieval system and decrease response times of individual requests.

First, let us assume that the entries of the data base are of fixed length. Two widely divergent methods of

access are applicable here. One is to construct a scatter index to secondary storage; the other is to organize the data base as a scatter storage table and access the data base directly.

Let us first consider the use of a scatter index table. Since accesses are made randomly there is no special advantage in locking any particular page in core; hence we suppose that the entire table is in secondary storage.

If we use either random probing or chaining as a collision-resolving discipline then we can expect the $(i+1)^{st}$ probe to be in the same page as the i^{th} probe with probability $\frac{1}{2^m}$ where 2^m is the number of pages that the table occupies. We must also allow one probe to get the item from the data base after its index is learned. The expected number of disc accesses for each discipline is therefore:

random probing

$$E_{rp} = 2 + \left(\frac{2^m-1}{2^m}\right)\left(-\frac{1}{\alpha}\ln(1-\alpha) - 1\right)$$

chaining

$$E_{ch} = 2 + \left(\frac{2^m-1}{2^m}\right)\left(\frac{\alpha}{2}\right)$$

where α is the table density.

If we use the next empty place method of resolving collisions we only access additional pages when the first probe lies near the end of a page. (Since even for a table density of .99 the expected number of probes is 50.5 and since a typical page size is 512 words we need not consider the possibility of accessing more than two index pages in order to calculate the expected number of accesses.)

next empty place

$$E_{ne} = 2 + \left(\frac{1}{2^n}\right)\left[\left(\frac{1}{2}\right)\left(\frac{2-\alpha}{1-\alpha}\right) - 1\right]$$

$$= 2 + \frac{1}{2^n+1}\left(\frac{\alpha}{1-\alpha}\right)$$

where 2^n is the page size.

Assuming an index of 256 pages of 512 words each we get the following table:

Table 1

α	.5	.6	.7	.8	.9
random probing	2.39	2.53	2.72	3.01	3.56
chaining	2.25	2.30	2.35	2.40	2.45
next entry place	2.00	2.00	2.00	2.01	2.01

From the table we can observe that even though the next

empty place method is computationally the least efficient it is nevertheless the most suitable as far as disc accesses are concerned.

We now propose a variation of the scatter index table which is computationally efficient and which requires no more disc accesses than the next empty place method. This method uses a computed hash code of $m+n$ bits. The first m bits are used to index a locked-in table of 2^m entries, each of which is a page address. The addressed page is brought into core, and the low order n bits are used as a key to locate the entry in this page. Any one of the three collision-resolving methods may be used. We emphasize that, whichever method is chosen, it is only applied within the one page (e.g., the pseudo-random number generator used in the random probing method generates integers between 1 and 2^n). Obviously, this method will not work if more than 2^n entries map into the same page. We now show that for acceptable table densities this overflow condition is so improbable that it may be safely ignored. Later we will present a method of avoiding overflow entirely.

Since hash addresses are assumed to be computed randomly, the probability of an entry being mapped into a given page is $\frac{1}{2^m}$. We have, in fact, a binomial distribution.

Writing p_k for the probability of k entries mapping into the same page, we have

$$p_k = \binom{\alpha N}{k}\left(\frac{1}{2^m}\right)^k \left(1 - \frac{1}{2^m}\right)^{\alpha N-k}$$

where $N = 2^{m+n}$ is the number of places in the table

α , as usual, is the table density

Now

(1) $$\Pr\,[\text{overflow}] = 1-\Pr\,[\text{no overflow}]$$
$$= 1-\sum_{k=0}^{2^n} p_k$$

We note that $(\alpha N\frac{1}{2^m})^2/2\alpha N$ is large in general, so that the Poisson approximation is not applicable. However, $\alpha N\frac{1}{2^m}(1-\frac{1}{2^m})$ is of reasonable magnitude (e.g., for the case of 256 512-word pages and $\alpha = \frac{1}{2}$, it is 255), and so we may employ the normal approximation. We then have[1]

(2) $$\Pr\,[\text{overflow}] \sim 1-\left[\Phi(x_{2^n+\frac{1}{2}}) - \Phi(x_{-\frac{1}{2}})\right]$$

---

[1]W. Feller, An Introduction to Probability Theory and its Applications, Vol. 1, page 172.

where $x_t = (t-\alpha 2^n)h$

$$h = [\alpha 2^n(1 - \frac{1}{2^m})]^{-\frac{1}{2}}$$

We present a table of these probabilities for the case of 256 512-word pages with various values of α

Table 2

$\alpha =$	.5	.6	.7	.8	.9
Pr[overflow]	$<8\times10^{-24}$	$<8\times10^{-24}$	1.8×10^{-16}	1.6×10^{-7}	.008

We observe that for $\alpha \leq .7$, it is so improbable that overflow occurs that the possibility may safely be ignored. Note that with this method exactly two accesses are required, one to retrieve a page of index table and one to retrieve a page from the data base.

We now discuss a method which eliminates the possibility of overflow. Even in cases of high table density, overflow is so improbable that relatively few pages are involved. We can eliminate overflow by splitting any page with 2^n entries into two parts and writing one part on a new page. An obvious way to do this would be to compute originally a hash code of $n+m+r$ bits, saving the r extra bits with each entry. Then, when a page

is split into two pages the low order bit is examined to determine in which of the two pages the entry is to go. Thereafter $m+1$ bits are necessary to determine which index page should be brought in from disc. Since m bits are used ordinarily the problem is to determine when m bits are insufficient. This is a table look-up problem amenable to scatter storage techniques. We create a small locked-in hash table (e.g., 64 entries for a table of 256 pages) using some of the m bits used to identify a page as a key. (Notation: we call this table a cluster-buster table.) Entries are made in the table whenever a page is split, each entry being the address of the new page which now contains part of the old page. When adding a new item or looking up an old one, reference is first made to the cluster-buster table. If no entry is found, then the page has never been split and its address can be found in the locked-in index table. If an entry is found, then the $(m+1)^{st}$ bit of the key is examined. If the bit is 0 then the first m bits of the key are an index to locate the page address in the locked-in index table. If the bit is 1, then the small hash table entry contains the page address.

Note that this procedure is reversible. That is, if sufficiently many deletions are made in the two halves of a split page, then the two halves may be recombined into

one page and the entry deleted from the cluster-buster table. Note also that this procedure, with some modification, may be applied to previously split pages which have again filled up. Hence it is possible to have pages in the index which have split several times, whereas other pages are only partially full. (In order to implement multiple splitting the following change must be made: the entry in the cluster-buster table contains (1) the number of additional bits needed to identify all the pages into which the original page was split; e.g., if a page is split into four pages then two additional bits, m+2 bits in all, are needed, and (2) the address of a locked-in auxiliary table, indexed by the additional bits, which contains the disc addresses of each page.) These two features (i.e., recombination and multiple splitting) allow a single index table to accomodate a varying skewed data base. In the event that additional locked-in space in core is available and that it is not expected that a page will ever have to be split more than once, the cluster-buster hash table can be replaced by an indexed table; this would use space inefficiently, but the time advantage of indexing as opposed to probing a hash table might justify the waste. For a data base of indeterminate size the cluster-buster technique can be used to trigger an overall system expansion. When the cluster-buster table has more than a certain percentage of entries

the entire system can be doubled. Those pages which have already split will not be split again and all entries for pages which have split exactly once can be deleted from the cluster-buster table.

Note that use of the cluster-buster table, just as ignoring improbable overflow, guarantees that precisely two disc accesses are required. We now investigate the disc access efficiency of each method. Since the collision resolving method within each page of index table does not affect the probability of overflowing that table, there is no reason not to use the most efficient method -- chaining. Similarly there is no reason not to use chaining within the cluster-buster table. Disc access cost is a function of the number of probes required to access an item. If k items are mapped into a page, then the expected number of probes to access an item on that page is $1 + \frac{k}{2^{n+1}}$. We have previously calculated that p_k, the probability that k items are mapped into a page, obeys the binomial distribution. Hence, ignoring overflow

$$E_{probe} = \sum_{k=0}^{2^n} p_k \left(1 + \frac{k}{2^{n+1}}\right)$$

$$\leq \sum_{k=0}^{\alpha N} p_k \left(1 + \frac{k}{2^{n+1}}\right)$$

$$= \sum_{k=0}^{\alpha N} p_k + \frac{1}{2^{n+1}} \sum_{k=0}^{\alpha N} p_k k$$

$$= 1 + \frac{1}{2^{n+1}} \alpha N \frac{1}{2^m}$$

$$= 1 + \frac{\alpha}{2}$$

The analysis of the cluster-buster case is somewhat more complicated. We will not consider the case where it is necessary to split a page more than once. The probability of such an event is so remote that, should it occur, it can be argued that the hash function is not acting randomly, in which event none of this analysis is valid anyway. We write p'_k for the probability that when a page is split into pages x and y, then page x has exactly k elements. We have

$$p'_k = \binom{2^n}{k}\left(\frac{1}{2}\right)^{2^n}$$

since entries from the split page go into page x with probability $\frac{1}{2}$. The expected number of probes in a page with k elements is $1 + \frac{k}{2^{n+1}}$.

Hence

$$\frac{1}{2}\left[(1 + \frac{k}{2^{n+1}}) + (1 + \frac{2^n - k}{2^{n+1}})\right]$$

is the expected number of probes to access an element which maps into a split page. Therefore the total contribution from all split pages is

$$E_1 = \Pr[\text{overflow}] \sum_{k=0}^{2^n} p'_k \, \frac{1}{2}\left[(1 + \frac{k}{2^{n+1}}) + (1 + \frac{2^n-k}{2^{n+1}})\right]$$

$$= \Pr[\text{overflow}] \frac{5}{4} \sum_{k=0}^{2^n} p'_k$$

$$= \frac{5}{4} \Pr[\text{overflow}]$$

The contribution from the unsplit pages (i.e., the pages where overflow has not occurred) is calculated as above.

$$E_2 = (1 - \Pr[\text{overflow}]) \sum_{k=0}^{2^n} p_k (1 + \frac{k}{2^{n+1}})$$

$$= (1 - \Pr[\text{overflow}])\left(\sum_{k=0}^{\alpha N} p_k (1 + \frac{k}{2^{n+1}}) - \sum_{k=2^n+1}^{\alpha N} p_k (1 + \frac{k}{2^{n+1}})\right)$$

$$\leq (1 - \Pr[\text{overflow}])\left(\sum_{k=0}^{\alpha N} p_k + \frac{1}{2^{n+1}} \sum_{k=0}^{\alpha N} k p_k - \frac{3}{2} \sum_{k=2^n+1}^{\alpha N} p_k\right)$$

$$= (1 - \Pr[\text{overflow}])(1 + \frac{\alpha}{2} - \frac{3}{2} \Pr[\text{overflow}])$$

Hence

$$E_{\text{probe}} = E_1 + E_2$$

$$\leq 1 + \frac{\alpha}{2} - \Pr[\text{overflow}](\frac{\alpha}{2} + \frac{5}{4} - \frac{3}{2} \Pr[\text{overflow}])$$

Figure XXVIII-1 is a graph of the term in parentheses as a function of α (using the normal approximation to calculate Pr[overflow] and assuming a table of 256 512-word pages). Note that this term is always positive. We see that if we either ignore improbable overflow or use a cluster-buster table, we require fewer probes than the most efficient collision-resolving method. To offset this saving we have for one method the possibility (admittedly improbable) of system blow-up because of overflow and for the other the expense of using the cluster-buster table. In either case we have the expense of accessing an indexed table.

Figure XXVIII-1

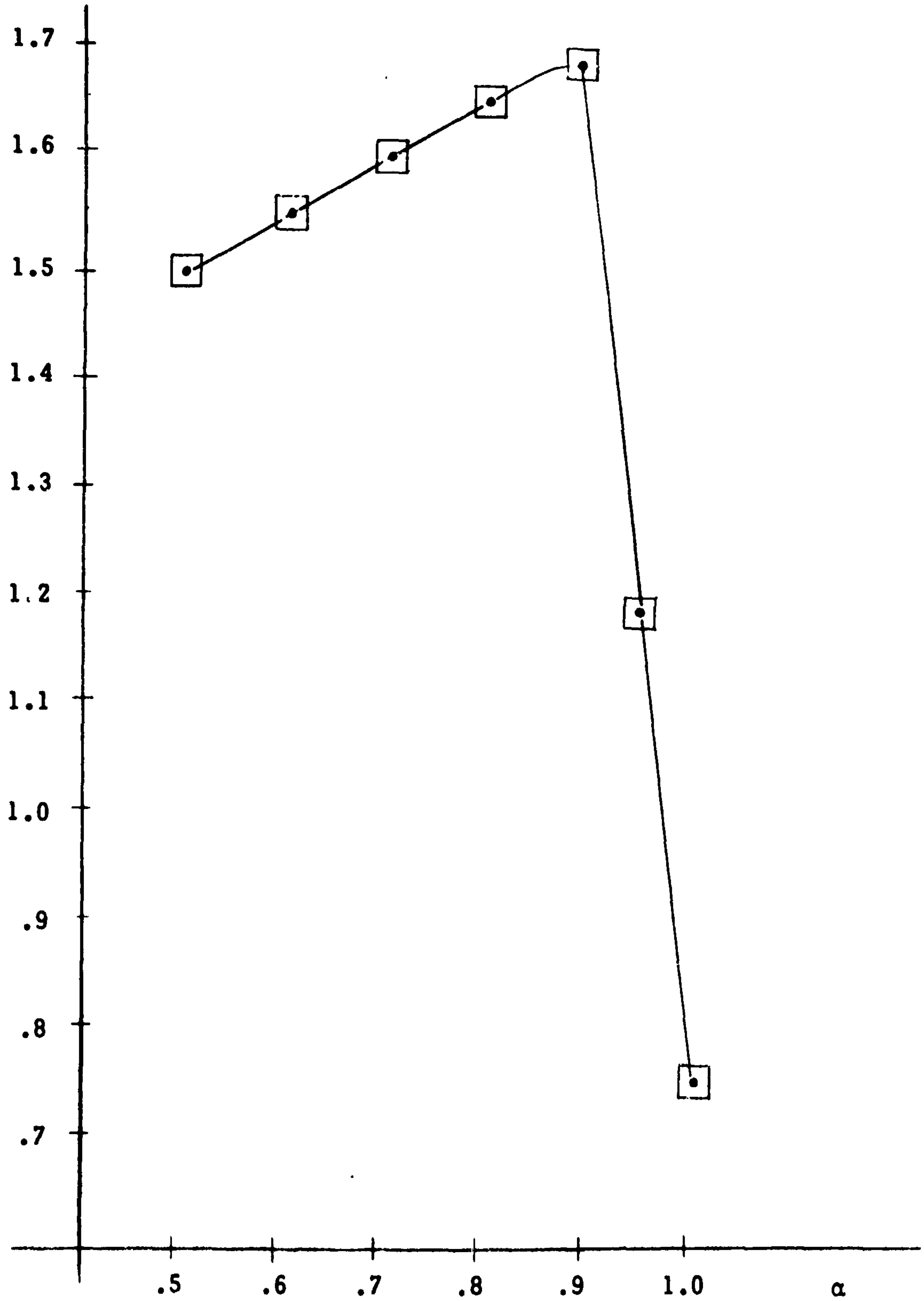
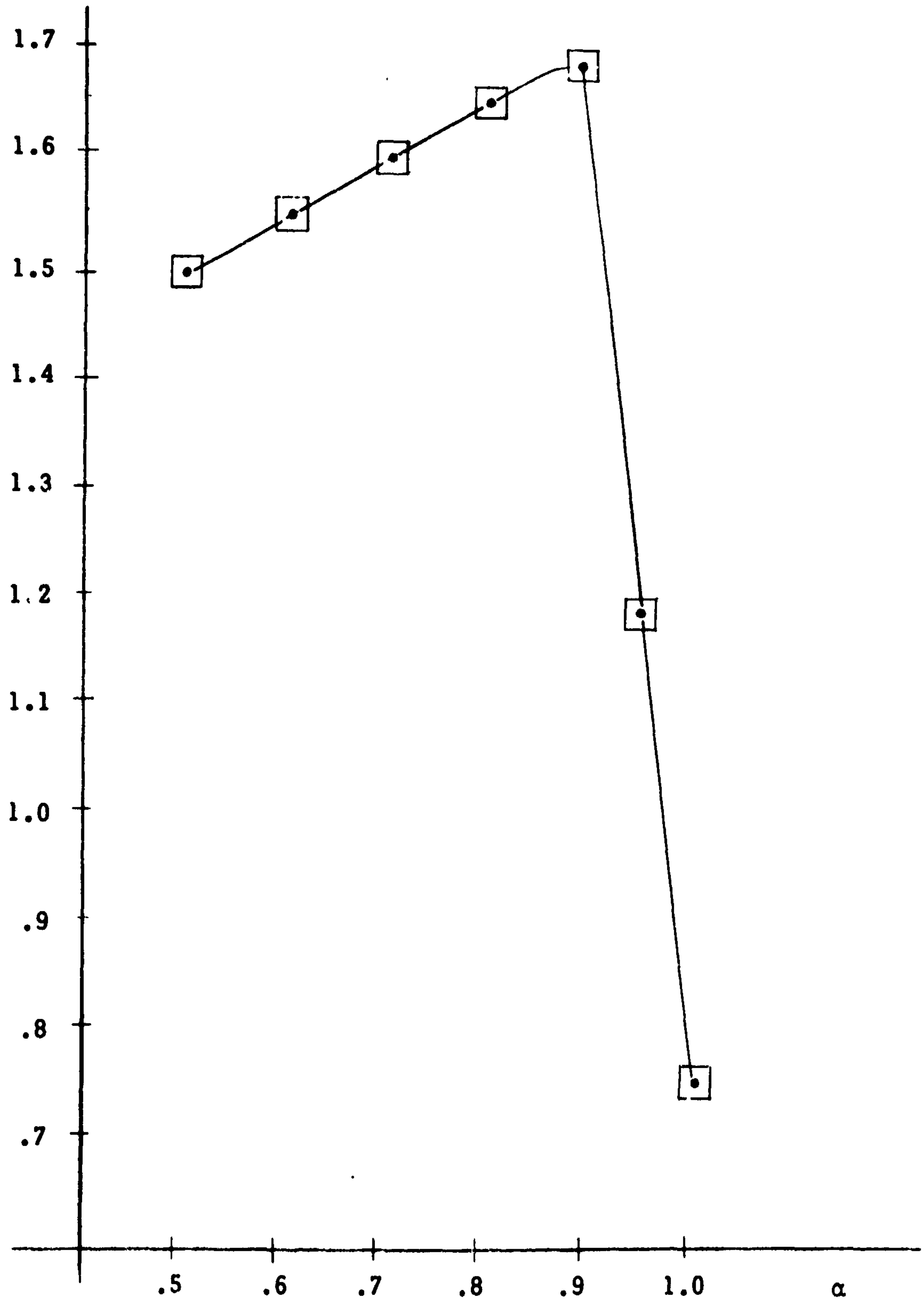


Figure XXVIII-1



In order to more readily compare all these methods we shall calculate computational costs (as opposed to costs of disc access which have already been presented). We define primitive costs as follows:

c = cost of compare

c_h = cost of computing a hash address $\cong$ 4c

c_x = cost of accessing a table through an index register $\cong$ c

c_r = cost of pseudo-random number generator $\cong$ 2c

c_{ch} = cost of following a chain $\cong$ 3c

c_i = cost of incrementing an address $\cong \frac{c}{2}$

Each of the following costs falls naturally into two parts: (1) the cost of choosing a disc page to bring into core; (2) the cost of locating an item within that page. The second of these costs is enclosed within {} in the following formulae.

c_1 = cost of random probe scatter index

$$= c_h + \{c_x+c+(c_r+c_x+c)(-\frac{1}{\alpha}\ln(1-\alpha)-1)\}$$

$$\cong 2c - 4c(\frac{1}{\alpha})\ln(1-\alpha)$$

c_2 = cost of chained scatter index

$$= c_h + \{c_x + c + (c_{ch} + c)(\tfrac{\alpha}{2})\}$$

$$\cong 6c + 2c\alpha$$

c_3 = cost of next empty place scatter index

$$= c_h + \{c_x + c + (c_i + c)\tfrac{1}{2}(\tfrac{\alpha}{1-\alpha})\}$$

$$\cong 6c + \tfrac{3}{4}c(\tfrac{\alpha}{1-\alpha})$$

c_4 = cost of paged scatter index ignoring overflow

$$= c_h + c_x + \{c_x + c + (c_{ch} + c)(\tfrac{\alpha}{2})\}$$

$$\cong 7c + 2c\alpha$$

c_5 = cost of paged scatter index using hashed cluster-buster filled to density $\frac{1}{2}$

$$\leq c_h + c_x + c + \Pr[\text{overflow}]\left[(c_h + c)\tfrac{1}{4} + c + \tfrac{1}{2}c_x\right]$$

$$+ (1 - \Pr[\text{overflow}])\left[(c_{ch} + c)\tfrac{1}{2} + c_x\right] + \{c_x + c + (c_{ch} + c)$$

$$\times \left[\tfrac{\alpha}{2} - \Pr[\text{overflow}](\tfrac{\alpha}{2} + \tfrac{5}{4} - \tfrac{3}{2}\Pr[\text{overflow}])\right]\}$$

$$\cong 9c + 3c\alpha + c\,\Pr[\text{overflow}](9\Pr[\text{overflow}] - 3\alpha - \frac{31}{4})$$

c_6 = cost of paged scatter index using indexed cluster-buster

$$\leq c_h + c_x + c + \Pr[\text{overflow}](c + \frac{1}{2}c_x)$$

$$+ (1 - \Pr[\text{overflow}])c_x + \{c_x + c + (c_{ch} + c)$$

$$\times \left[\frac{\alpha}{2} - \Pr[\text{overflow}](\frac{\alpha}{2} + \frac{5}{4} - \frac{3}{2}\Pr[\text{overflow}])\right]\}$$

$$\cong 9c + 2c\alpha + c\,\Pr[\text{overflow}](6\Pr[\text{overflow}] - 2\alpha - \frac{9}{2})$$

These formulae are, of course, only approximations and can easily be obtained by considering the operations that are necessary to access an entry by each method. Note that they compute the cost of accessing an entry which is in the table. The cost of trying to access an item not in the table is higher. The following table gives these costs as a function of α for the case of a table of 256 512-word pages.

Table 3

α =	.5	.6	.7	.8	.9
c_1/c	7.54	8.11	8.88	10.05	12.23
c_2/c	7.00	7.20	7.40	7.60	7.80
c_3/c	6.75	7.13	7.75	9.00	12.75
c_4/c	8.00	8.20	8.40	8.60	8.80
c_5/c	10.50	10.80	11.10	11.40	11.62
c_6/c	10.00	10.20	10.40	10.60	10.75

In most systems, however, disc access time is of greater concern than computational cost. In that case, ignoring overflow or using a cluster-buster is the best method, with the next empty place method close behind. We now consider the costs involved in organizing the data base as a hash table and accessing items directly. Items in the data base are assumed to occupy 2^r words, $0 \leq r \leq n$. We require 2^{m+r} pages to store as many items in a scatter storage table as could be accessed by 2^m pages of a scatter index table of the same density. We note immediately that only values of α close to 1 can be considered, unless we are willing to waste huge amounts of secondary storage (e.g., for $r = 4$, $m = 8$, $n = 9$, and $\alpha = .8$, we waste approximately 51.2 pages (27K words) using a scatter index table and 819.2 pages (410K words) using a scatter storage table. If items are larger, and

in real use (e.g., inverted lists in an information retrieval system) they often will be, the waste in a scatter storage table at any density not close to 1 becomes intolerable.) If we use either random probing or chaining, the probability of the $(i+1)^{st}$ probe being in the same page as the i^{th} probe is $\frac{1}{2^{m+r}}$. Hence the expected number of accesses required is:

random probing

$$E_{rp} = 1 + \frac{2^{m+r}-1}{2^{m+r}}\left(-\frac{1}{\alpha}\ \ln(1-\alpha)-1\right)$$

chaining

$$E_{ch} = 1 + \frac{2^{m+r}-1}{2^{m+r}}\left(\frac{\alpha}{2}\right)$$

If the next empty place method is used we must use a more careful analysis. Let $E = \frac{1}{2}\left(\frac{2-\alpha}{1-\alpha}\right)$. Then $\left\lceil \frac{E}{2^{n-r}} \right\rceil$ is the minimum number of disc accesses required. Now $(E-1) - \left\lfloor \frac{E-1}{2^{n-r}} \right\rfloor 2^{n-r}$ is the remainder of $E-1$ divided by 2^{n-r} and hence $\dfrac{(E-1) - \left\lfloor \frac{E-1}{2^{n-r}} \right\rfloor 2^{n-r}}{2^{n-r}}$ is the probability that this remainder will overflow a page boundary, i.e., the probability that 1 more than the minimum number of disc accesses will be required. Hence

next empty place

$$E_{ne} = \left\lceil \frac{E}{2^{n-r}} \right\rceil + \frac{(E-1) - \left\lfloor \frac{E-1}{2^{n-r}} \right\rfloor 2^{n-r}}{2^{n-r}}$$

$$= \left\lceil \frac{E}{2^{n-r}} \right\rceil - \left\lfloor \frac{E-1}{2^{n-r}} \right\rfloor - \frac{E-1}{2^{n-r}}$$

$$= 1 + \frac{E-1}{2^{n-r}}$$

$$= 1 + \left(\frac{1}{2^{n-r+1}}\right)\left(\frac{\alpha}{1-\alpha}\right)$$

Note that this is precisely the same formula as derived above where we discounted the possibility of a search going over more than one page boundary.

We now present a table of these expected numbers of accesses for the case of $m = 8$, $n = 9$, $\alpha = .9$, and $r = 4 , 5 , 6 , 7$

Table 4

$r =$	4	5	6	7
E_{rp}	2.56	2.56	2.56	2.56
E_{ch}	1.45	1.45	1.45	1.45
E_{ne}	1.14	1.28	1.56	2.13

From Tables 2 and 4 we observe that if entries in the data base are of a fixed length of 32 words or less, and

if it is possible to waste a substantial piece of secondary storage (for the case $r = 4$, 205K words are wasted), organizing the data base as a scatter storage table using the next empty place method of collision-resolution is most efficient with respect to number of disc accesses required. If it is not possible to waste as much space, or if entries are larger than 32 words, then organizing the data base as a scatter storage table using chaining is most efficient. (If $\alpha = .99$ the wasted space drops to 20K words while $E_{ch} = 1.50$.)

We must note that organizing the data base as a scatter storage table is possible only if entries are of fixed length. In general, unfortunately, entries are of variable unbounded length, e.g., text, inverted lists, item-sequenced lists, etc. In that case, of the methods discussed so far, ignoring overflow or using a cluster-buster table are most efficient with respect to number of disc accesses, each method requiring exactly 2.

We now discuss a method which combines features of both scatter storage and index tables and which reduces the number of disc accesses. We retain a scatter index table, but instead of using the entire page for hash table, we devote a portion to entries from the data base. Since the hash segments are small, we can expect overflow to be fairly probable, so it cannot be ignored; therefore

we use a cluster-buster table to prevent overflow. It is possible to describe an organization where the proportion of each page devoted to data base items varies from page to page. However, in this discussion we shall only consider the case where this proportion is fixed. A hash table entry will either point to an entry in the page or will be the disc address of the entry (again, other organizations are possible, but this is the only one we discuss here).

The proportion of each page devoted to data base entries and the density of the scatter index table depend on several conflicting criteria. One, it is desirable that as little data base storage space as possible be wasted. Hence it is desirable that there be sufficiently many entries in the hash table in each page so that the remainder of the page be filled: i.e., either α should be large, or the hash table section large, or both. Two, since disc accesses are expensive, as many items as possible should be in the page: i.e., the hash table section should be small. Three, core space is expensive and the cluster-buster resides in core. Hence, as few segments as possible of hash table should overflow: i.e., either α should be small, or the hash table section large, or both.

¶ We now present formulae for these criteria.

$$S = \text{size of cluster-buster table.}$$

$$\cong 2\Pr[\text{overflow}]\frac{N}{T}$$

where $\Pr[\text{overflow}]$ is given exactly by (1) and approximately by (2) above

N = number of places in scatter index table

T = number of places in each segment of table

W = space wasted in mixed pages

$$\cong [(2^n - T) - \alpha T \ell']\frac{N}{T}$$

where $\ell' = \sum_{i=0}^{2^n - T} ip_i \leq \bar{\ell}$ = expected length of each data base entry

p_i = probability that a data base entry is i words long

E = expected number of disc accesses required

$$= 1 + [1 - (\frac{1}{\alpha T})(\frac{2^n - T}{\ell'})$$

$$= 2 - \frac{2^n - T}{\alpha T \ell'} < 2$$

In sum, we have discussed various solutions to the problem of accessing a data file too large to fit into

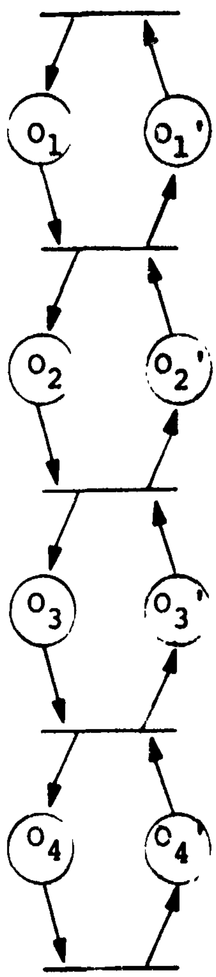
core. If the file consists of fixed size items, then organizing the data as a hash storage table and addressing it directly is most economical with regard to disc accesses. If the file consists of variable length items, then use of a mixed scatter index table with an auxiliary cluster-buster table is most economical.

XXIX. Net Models -- Some Elementary Constructs

In previous sections we have been satisfied with an informal definition of batching and buffering, and we have ignored the general question of concurrency -- despite the fact that many of the systems examined have involved concurrent operation. For example, when a subdeck of feature cards is placed in front of a light source, the "hits" are available concurrently; the intersection operation occurs concurrently for all card-positions. We will introduce Petri nets as a representational medium for exhibiting concurrency. A brief description of Petri nets and occurrence systems is provided in Appendix I.

Let us begin our discussion by considering a simple system consisting of four operations. When the first operation is completed, the second operation begins; when the second is completed, the third begins; when the third is completed, the fourth begins. These four operations, thus constrained, are repeated cyclically. We represent this system with the net in Figure XXIX-1. The sequencing constraints seem to preclude performing any of the operations concurrently. If this were the case, the time required for each iteration of the cycle would be equal to the sum of the durations of the four operations.

Figure XXIX-1



O_1 : operation 1 in progress

O_2 : operation 2 in progress

O_3 : operation 3 in progress

O_4 : operation 4 in progress

O_1' : operation 1 not in progress
(i.e., completed; not yet rebegun)

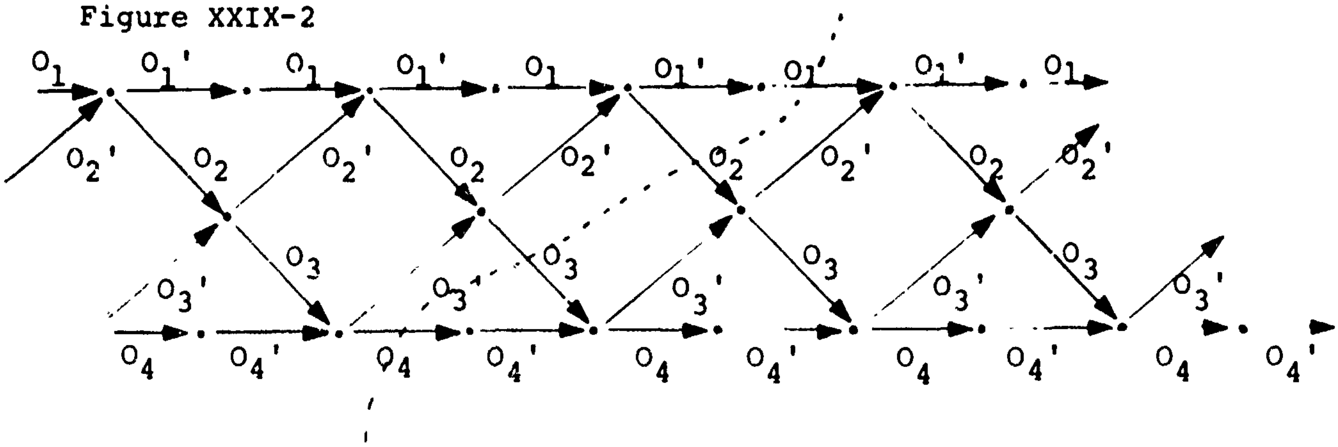
O_2' : operation 2 not in progress

O_3' : operation 3 not in progress

O_4' : operation 4 not in progress

However, Figure XXIX-2, which is a repetition stretch representing four iterations of the system's behavior-cycle, exhibits the fact that all four operations may be performed concurrently.

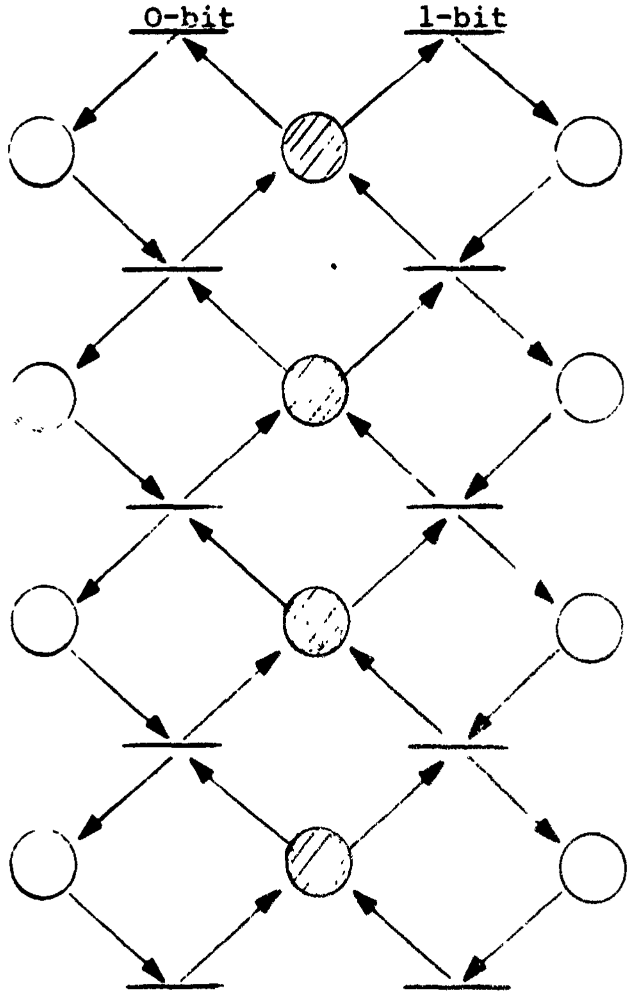
Figure XXIX-2



The net, which represents explicitly both the sequencing constraints and the cyclic behavior of the system, exhibits concurrencies among operations from (what we think of as) different iterations of its behavior-cycle. The dotted line in Figure XXIX-2 indicates a time slice in which the n^{th} iteration of operation 1, the $n-1^{st}$ iteration of operation 2, the $n-2^{nd}$ iteration of operation 3, and the $n-3^{rd}$ iteration of operation 4 are concurrent.

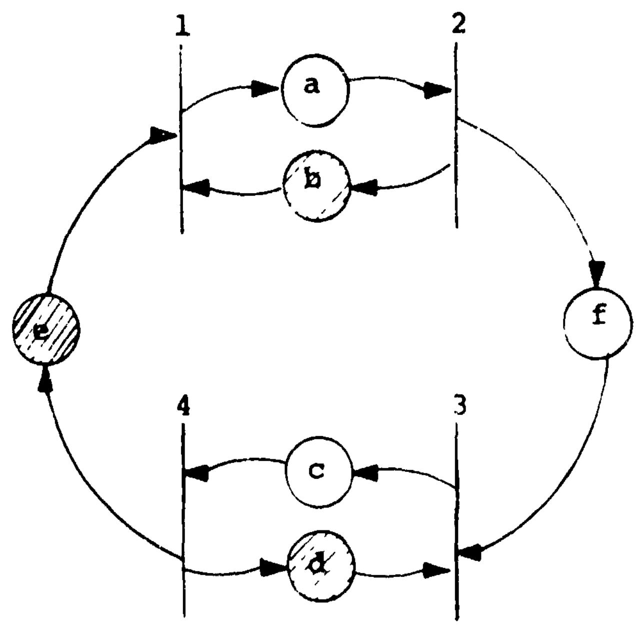
We shall call the net structure in Figure XXIX-1 a pipeline. The pipeline in Figure XXIX-1 has four stages; accordingly, we would describe it as a pipeline with capacity 4. We can view a pipeline variously as a set of ordered operations (as in the example above) or as a buffer or stack into which values can be placed. In the latter interpretation each "stage" or "pair of places" might be viewed as a storage cell capable of storing one value. We may think of values as being "dropped in at the top" and transmitted "down" the pipeline. A pipeline of capacity n will be capable of holding n values concurrently. Suppose we were dealing with two different types of value and we wished to distinguish between them. We could construct a bi-valued pipeline, or bit channel, as in Figure XXIX-3.

Figure XXIX-3



Each stage now has three possible (mutually exclusive) states: "empty", "1", or "0". Note that with such a structure we can transmit a sequence of bits, maintaining the order of the values.

XXX. A Model of Buffering



transition 1 : initiation of the process

transition 2 : termination of the process

transition 3 : initiation of buffered I/O

transition 4 : termination of buffered I/O

place a : process in progress

place b : process not in progress

place c : I/O in progress

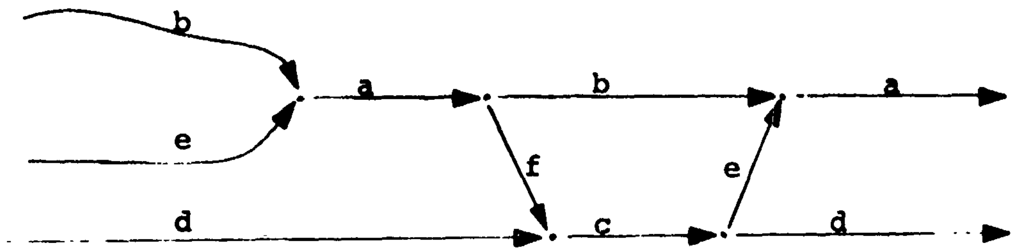
place d : I/O not in progress

place e : input for process available

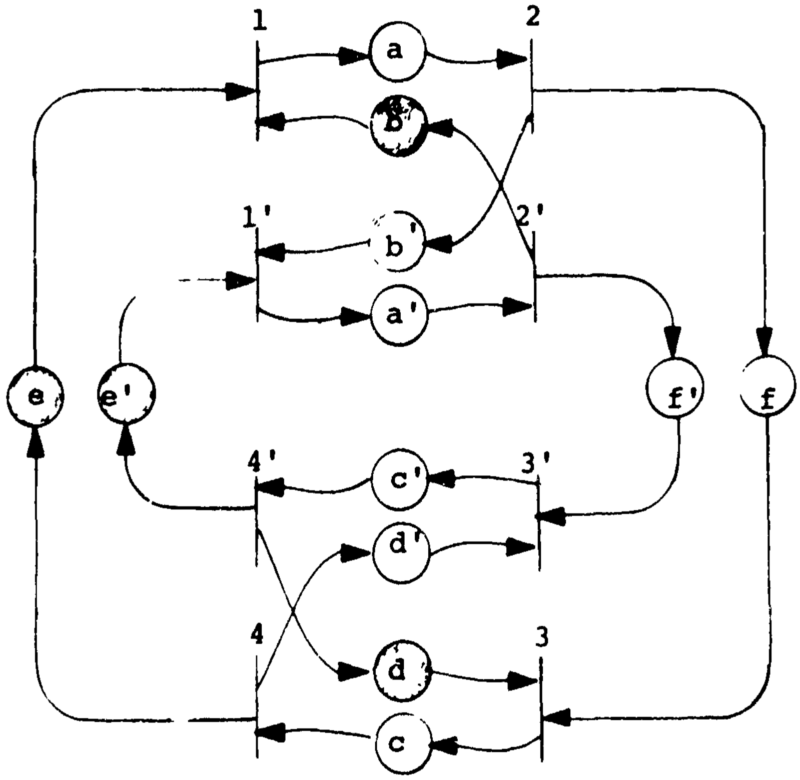
place f : output of process available for I/O

The occurrence graph below is based on an initial case in which both the process and the I/O are idle and the

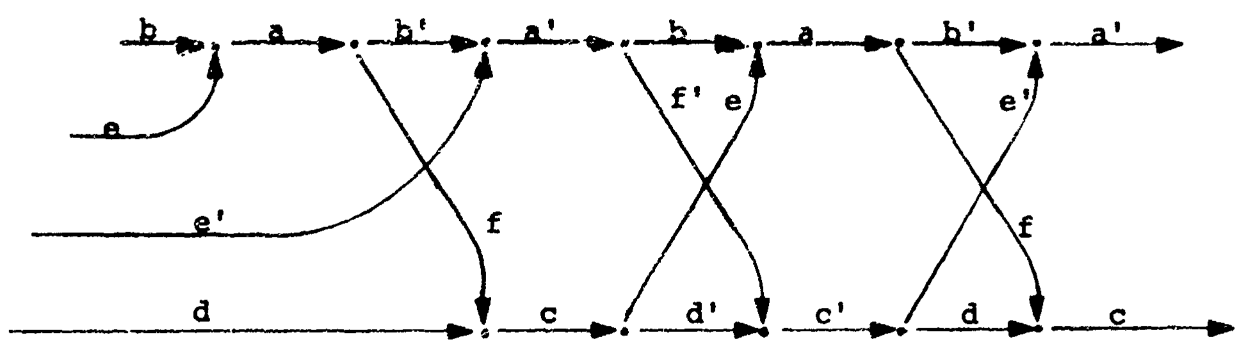
"input-buffer" of the process is full. Note that in this system the process and the I/O will never be concurrent. Thus, if a and c are the only places of significant duration, the minimum time for a cycle in this system is equal to a+c -- i.e., processing time + I/O time.



XXXI. A Model of Double Buffering



The labels correspond to those used in the model of single buffering, with primes added to show how the double buffer model is composed of two single buffers. Note that places b and b' and places d and d' guarantee alternation of transitions 1 and 1' and of transitions 3 and 3'.



This occurrence graph is based on an initial stage in which both the process and I/O are idle and both "input" buffers are loaded. In this occurrence graph the process and I/O are concurrent. Hence, if a (and a') and c (and c') are the only places of significant duration, the minimum time for a cycle in this system is max (processing time, I/O time).

The buffering model is generalizeable to any number of buffers. This becomes especially interesting when I/O time is greater than processing time and it is possible to perform several I/O operations concurrently. Consider the case of n concurrently operable I/O devices. Then minimum cycle time would be $\sim$ max (processing time,

$\frac{1}{n} \times$ I/O time). The dual also holds: if processing time is greater than I/O time and an execution of the process is not dependent on previous executions, the availability of m concurrently operable processors permits minimum cycle time to be $\sim$ max ($\frac{1}{m} \times$ processing time, I/O time).

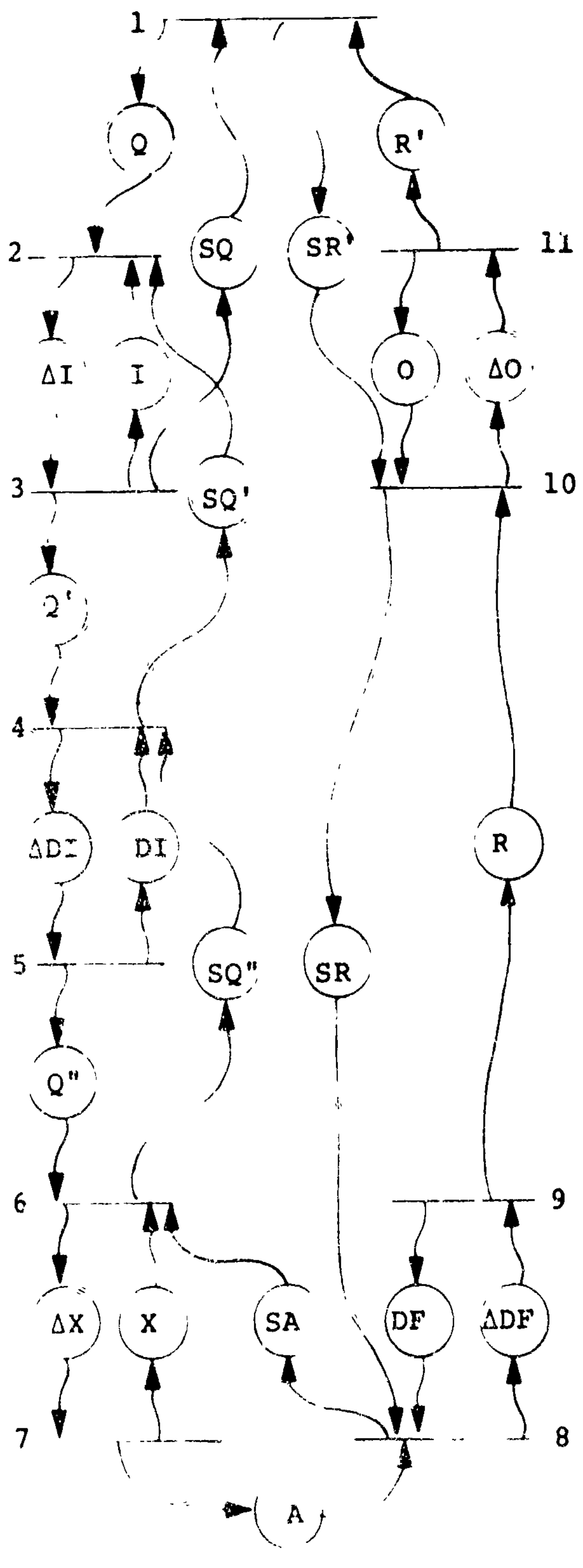
XXXII. Pipelined and Serial Phased Systems

In this section we will illustrate the distinction between pipelined phased systems and conventional serial phased systems, by comparing two systems which perform the same task.

Both systems perform as follows:

previous response is accepted next query is generated	:	event 1 (and event 1' for pipelined system)
query is input	:	event 2 initiates input; event 3 completes input
query is decoded	:	event 4 initiates decoding; event 5 completes decoding
cross-indexing accomplished	:	event 6 initiates cross-indexing event 7 completes cross-indexing
file access performed for hits	:	event 8 initiates file access event 9 completes file access
response records output	:	event 10 initiates output; event 11 completes output

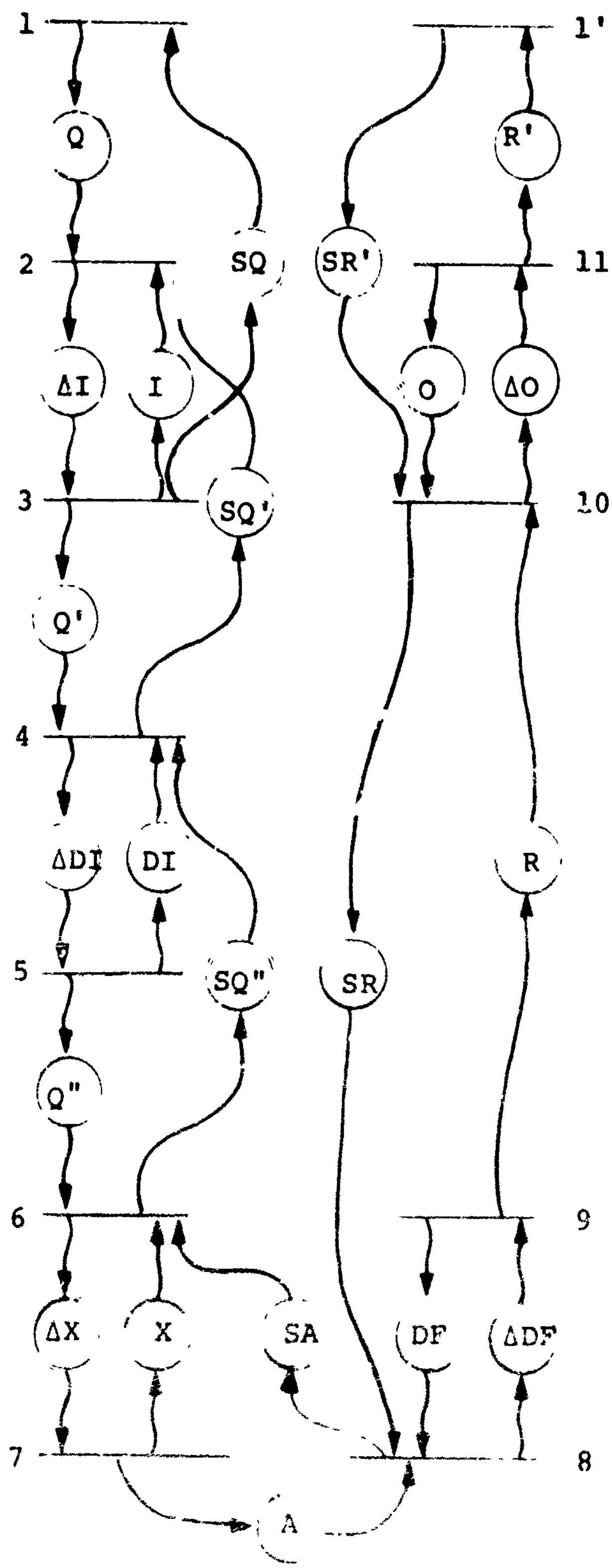
Serial Phased



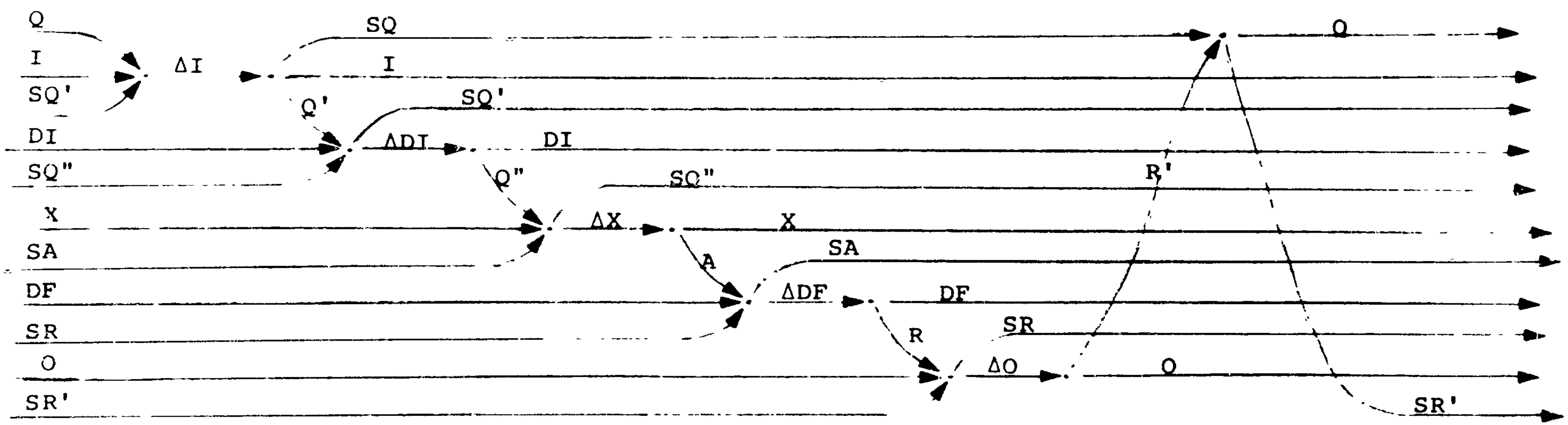
Q : query available
I : input channel available
ΔI : input in progress
Q' : internal query available
DI : dictionary available
ΔDI : decoding in progress
Q" : query decoded
X : cross-index available
ΔX : indexing in progress
A : accession numbers of hits available
DF : document file available
ΔDF : documents being retrieved
R : result available for input
O : output channel available
ΔO : output in progress
R' : result outputted

SQ : space for query, external
SQ' : space for query, internal
SQ" : space for query, decoded
SA : space for 'hits'
SR : space for documents retrieved
SR' : external space for documents

Pipeline Phased



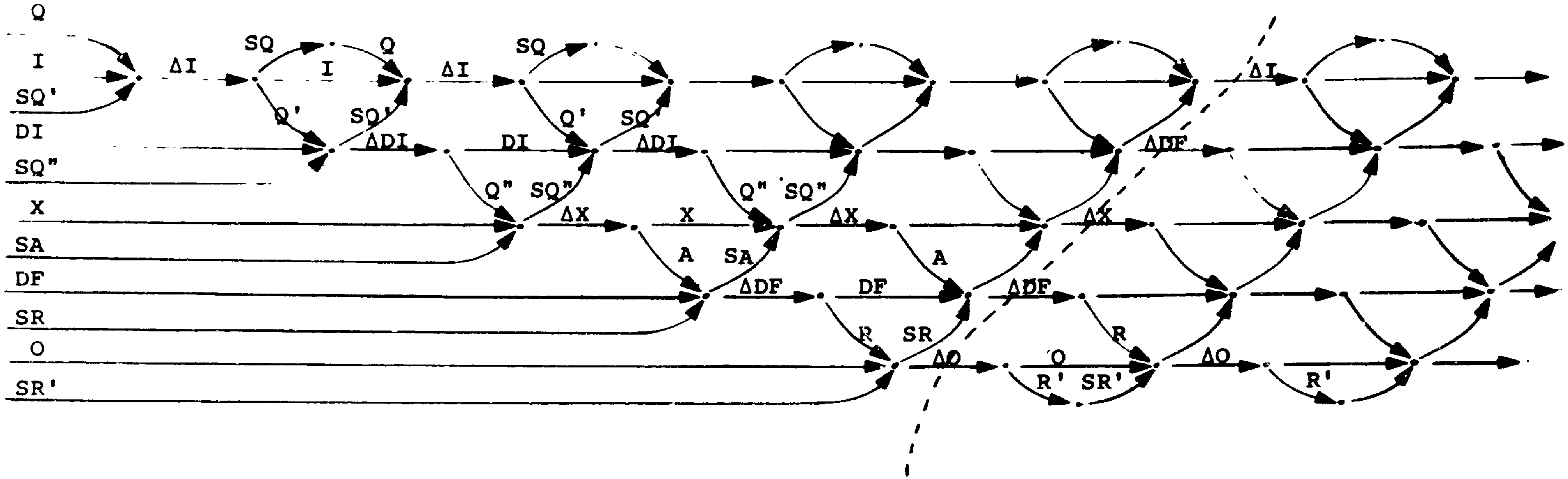
Serial Phased



If ΔI , ΔDI , ΔX , ΔDF , ΔO are the only places of significant duration, the minimum time for a cycle in this system is

(input time + decoding time + indexing time + document retrieval time + output time)

Pipeline Phased



The minimum time for a cycle in this system is

max (input time, decoding time, indexing time, document retrieval time, output time)

Notice that a cycle relates to throughput capacity: in both systems the time to process one query is the same. The serial system requires receiving the response to the first query before submitting the next. The pipeline phased system can process a number of queries concurrently. The processing stages are staggered in the pipeline. The cycle time is thus a measure of the maximum rate at which queries can be processed.

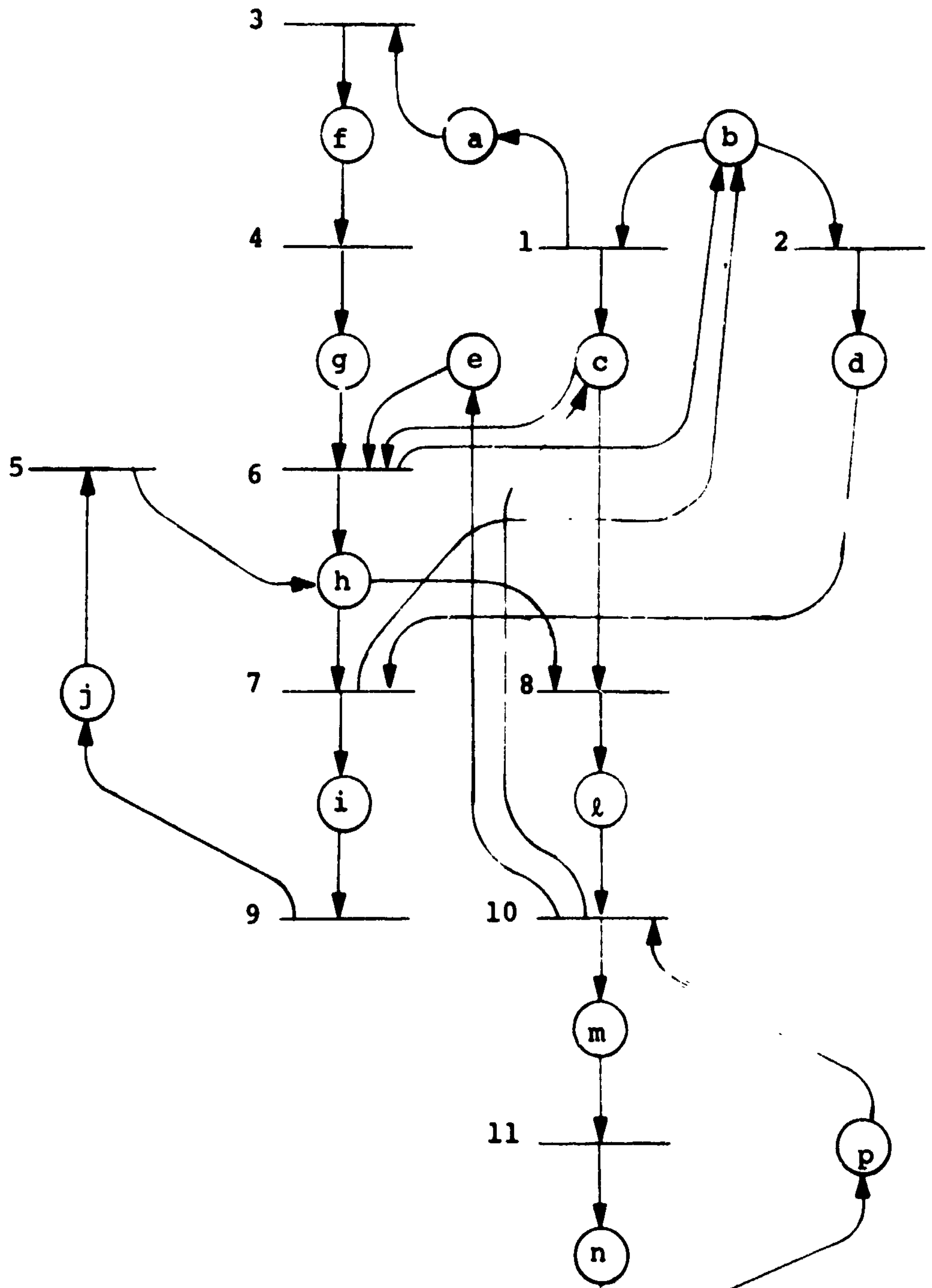
XXXIII. A Model of a Hardware Device -- The NCR CRAM Unit[1]

In this section we illustrate the use of Petri nets in modelling the synchrony and concurrency characteristics of pseudo-random access mass storage devices. The NCR CRAM Unit is a pseudo-random access mass memory device. The storage medium is 256 oxide-coated cards. Each card has a set of notches at one end, which permits the selection, at random, of any one of the cards from the CRAM magazine. When loaded into a CRAM unit, the cards hang from eight rods which may be turned in such a way as to release exactly one card. When the card is released, it falls freely until it reaches a rotating drum to which it is pulled by means of a vacuum, and the card is accelerated to the surface speed of the drum. Shortly after attaining this speed, the leading edge of the card reaches the read-write heads. After reading or writing, the card may remain on the drum, to be recirculated past the heads on the next revolution, or it may be released and returned to the magazine.

Three photocells provide the prime source of control of the mechanism. PE 1 is located between the return chute and the magazine and controls the operation of the loader

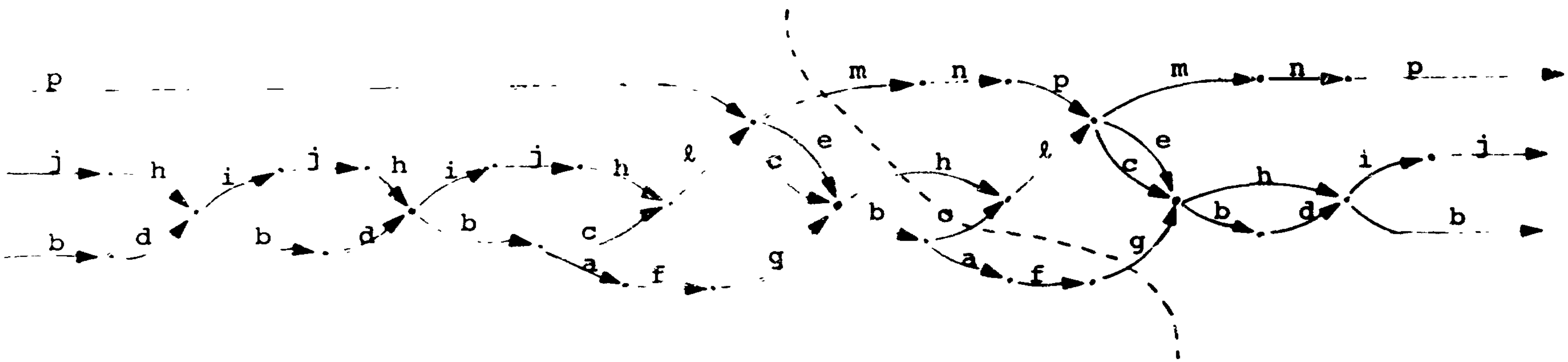
[1] See National Cash Register Company publication MD 315-101 10-62 for a description of the equipment.

mechanism for the magazine. PE 2 signals the arrival of a card, either one which has just been dropped, or one which is recirculated on the drum. Reading or writing must be done before the leading edge of the card arrives at PE 3. Here we focus on certain characteristics of the CRAM unit: we have modelled the elements in CRAM that relate to the seek time, and have not modelled the details of writing or reading, nor the reject mechanism which automatically rejects a card from the drum after 750ms. of total inactivity. We also have not modelled those phenomena associated with individual card identity (for instance, if you happen to select a card that was just ejected from the drum, there may be an additional time delay -- the length of time needed for the card to return to the magazine).


3
f
a
b
4
1
2
g
e
c
d
5
6
h
7
8
j
i
ℓ
9
10
m
11
p
n
12

1. Decision to select a card
2. Decision not to select a card. (When a card is on the drum, reject is triggered by the next card selection. Being passive, i.e., not issuing a select, means deciding not to select a card for each drum revolution that occurs with a card on the drum)
3. Beginning of select
4. End select, begin card drop
5. End (gate-PE 2), begin (PE 2-PE 3)
6. End card drop, begin (PE 2-PE 3) (Events 5 and 6 represent the two alternate routes by which a card passes the read-write station: Event 5 represents recirculation, Event 6 represents new arrival from magazine
7. End (PE 2-PE 3), begin (PE 3-gate) (recirculate)
8. End (PE 2-PE 3), begin (PE 3-gate) (eject)
9. End (PE 3-gate), begin (gate-PE 2) (recirculate)
10. End (PE 3-gate), begin (gate-PE 1) (eject)
11. End (gate-PE 1), begin (PE 1-magazine)
12. End (PE 1-magazine)

a decision made to select, select not yet begun
b no decision yet for this drum revolution
c decision made to select, card has not yet fallen
d decision made <u>not</u> to select
e no card on drum
f selection occurring
g card dropping
h leading edge between PE 2 and PE 3
i leading edge between PE 3 and gate
j card being recirculated
ℓ leading edge between PE 3 and gate
m leading edge between gate and PE 1
n card entering magazine
p no card in return chute



The initial case of the o-graph represents: a card circulating on the drum (j)
the return chute empty (p)
no decision yet made for this drum revolution (b)

This history shows the card staying on the drum for two revolutions, another card being selected, and yet another card being selected as soon as possible after that.

The time slice shows the following three conditions holding concurrently:

a card in the return chute (m)
a card on the drum where it can be read or written (h)
a card falling (g)

XXXIV. A Highly Concurrent Net Model of the Cross-Indexing Grid

In Section I we presented a model of cross-indexing information. The model consisted in a grid: horizontal lines represented items, and vertical lines represented descriptors; a given intersection j,k was circled if and only if descriptor j applied to item k . Note, however, that this representation is static -- it represents the cross-indexing information (i.e., the set of descriptor-item relations) used in performing a query (or update), but it does not represent the actual process of perf-rming a query (or update). In this section, then, we will develop a Petri net grid model of cross-indexing which represents the process of query performance. (It will become clear that the model can be expanded to represent the performance of updates.) This model will exhibit possibilities for concurrency which may be exploited by batching, buffering, or pipelining.

In Section XXIX we introduced a net mc el of a bit channel. Roughly speaking, the net model of the cross-indexing grid is constructed by replacing each horizontal and each vertical line in the grid with such a bit channel. We may think of the vertical channels as transmitting upward and of the horizontal channels as transmitting from right to left. A query is made by selecting a subset of the descriptors. In the net model, then, a query will be made

by supplying a value to each of the vertical channels as follows: a "1" is supplied if the corresponding descriptor is in the query set; a "0" is supplied if it is not. As a value is transmitted up a vertical channel, a "copy" of it is "deposited" at each intersection. Furthermore, at each intersection j,k a value is already stored as follows: a "1" if descriptor j applied to item k (i.e., if the intersection is circled); a "0" if not. Each of the horizontal channels generates "1's" continuously from its right end and transmits them leftward; as a "1" is transmitted leftward it may be transformed into a "0" as a function of the state of one of the intersections it encounters. If the value which reaches the left end of a given horizontal channel is a "1", then the corresponding item satisfies the query: if it is a "0", the item does not satisfy the query. Let us call the bit stored at a given intersection the "cross-indexing bit", the bit received from the vertical channel the "query bit", the bit received from the horizontal channel the "incoming response bit", and the bit transmitted leftward (to the next intersection) the "outgoing response bit".

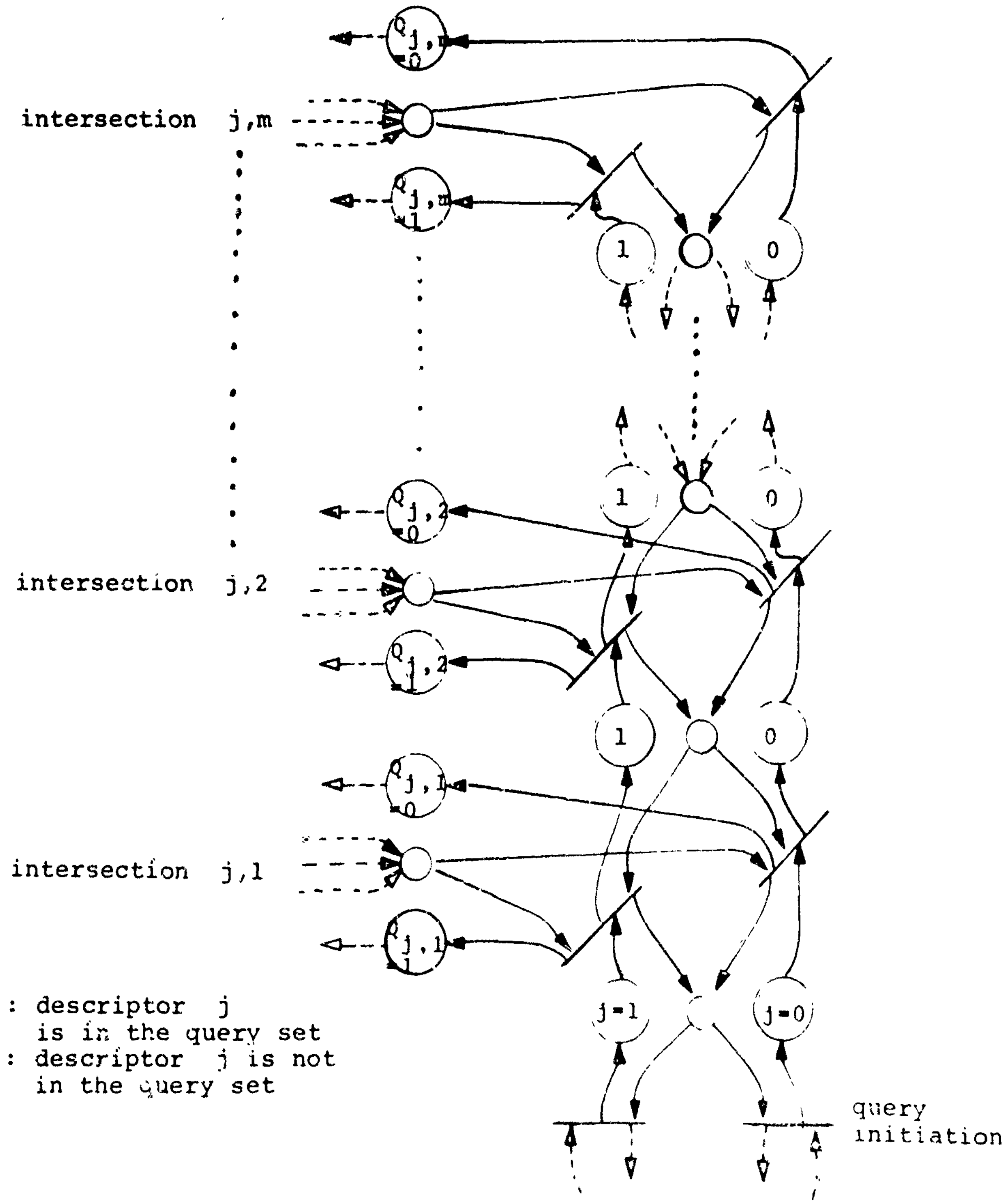
We can then describe the logic at an intersection j,k as follows: if the incoming response bit is a "0", then the outgoing response bit will be a "0" (i.e., it has already been determined that item k does not satisfy

the query); if the incoming response bit is a "1" and the query bit is a "0", the outgoing response bit is a "1" (i.e., descriptor j is not in the query set); if the incoming response bit is a "1" and the query bit is a "1" and the cross-indexing bit is a "0", then the outgoing response bit is a "0" (i.e., j is in the query set and it does not apply to k); if the incoming response bit, the query bit, and the cross-indexing bit are all "1's", then the outgoing response bit is a "1" (j is in the query set and applies to k). An additional vertical channel is provided at the left edge of the grid model for "reading out" the results of a query.

Because the various elements of the net model intersect each other, we will not try to represent an entire net grid pictorially. Instead we present the elements individually below:

Figure XXXIV-1

Vertical Channel for Descriptor j. The value of descriptor k for each query is "tapped off" at each intersection.



j=1 : descriptor j is in the query set
j=0 : descriptor j is not in the query set

Figure XXXIV-2

Horizontal Channel for Item k

R_k=1 : item k satisfies the query

R_k=0 : item k does not satisfy the query

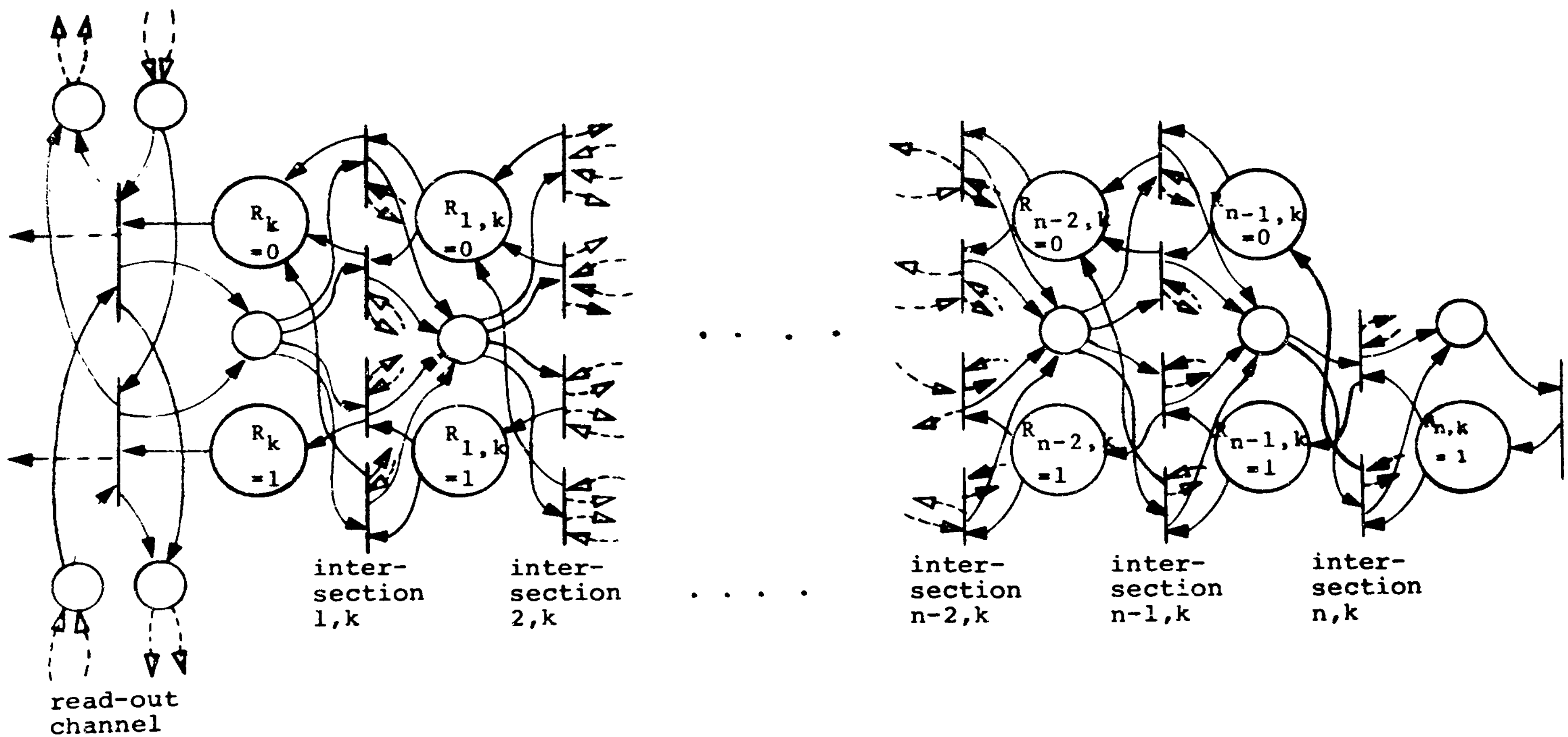


Figure XXXIV-3

Intersection j,k

$R_{j,k}=0$: it is already known that item k does not satisfy the query

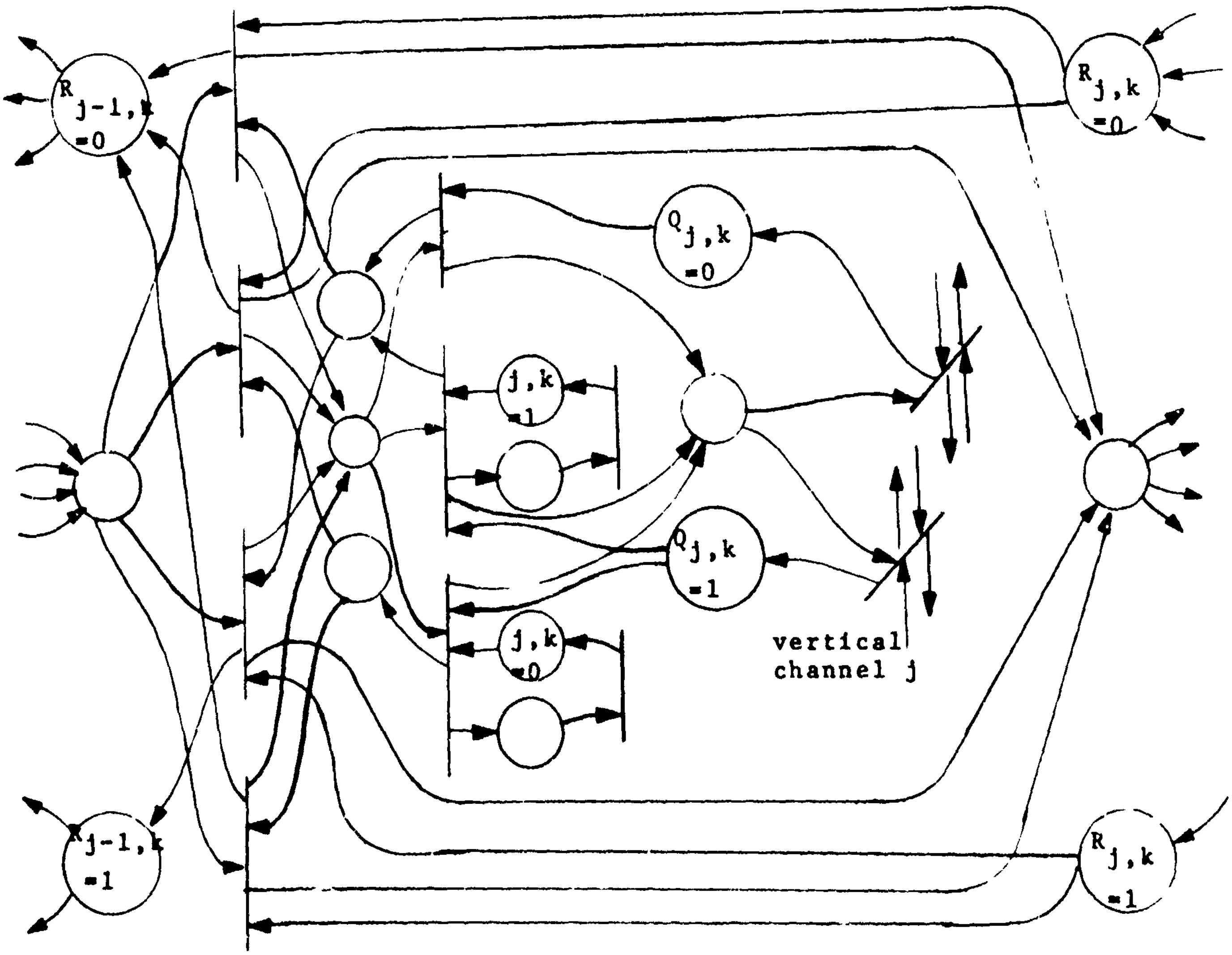
$R_{j,k}=1$: thus far item k satisfies the query

$Q_{j,k}=0$: descriptor j is not in the query set

$Q_{j,k}=1$: descriptor j is in the query set

j,k=0 : descriptor j does not apply to item k

j,k=1 : descriptor j applies to item k



Note that the elements of our net model of the cross-indexing grid are pipelines so that it is capable of highly concurrent operation. The number of queries which can be evaluated concurrently is equal to I+D , where I = the number of items in the system and D = the number of descriptors in the system. The processing time for one query -- i.e., the time between initiation of a given query and completion (i.e., response read-out) of that query -- will be equal to C(I+D) , where C is a constant. However, the throughput rate will be equal to C . That is, if queries can be input at a sufficient rate (i.e., approaching C), the time between successive outputs will approach C .

APPENDIX I

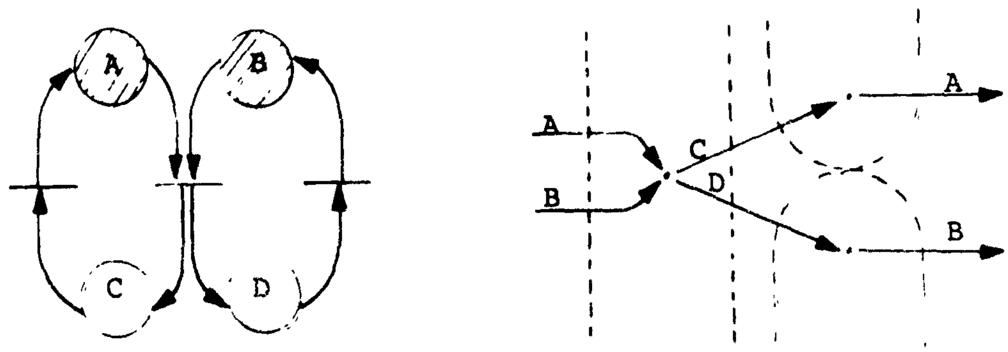
Petri Nets[1]

Formally, a Petri net is a directed graph with two kinds of nodes: places, represented as circles; and transitions, represented as line segments. Each directed arc, represented as an arrow, connects one place with one transition. An arrow from a place to a transition means that the place is an input to the transition; an arrow from a transition to a place means that the place is an output of the transition. Every place in a net is an output of at least one transition and an input to at least one transition. No place may be both an input to and an output of the same transition. A place is capable of two states: full or empty. The state of a net is given by a list of all its full places. A transition may fire if and only if all of its inputs are full. When a transition fires, all of its inputs are emptied and all of its outputs are filled. If some place is input to two or more transitions, all of whose inputs are full, these transitions are in conflict. Only one of the transitions -- any one -- may fire in such a situation. (See Figures A, B, and C for examples of net diagrams. Figure B shows a net with conflict.)

---

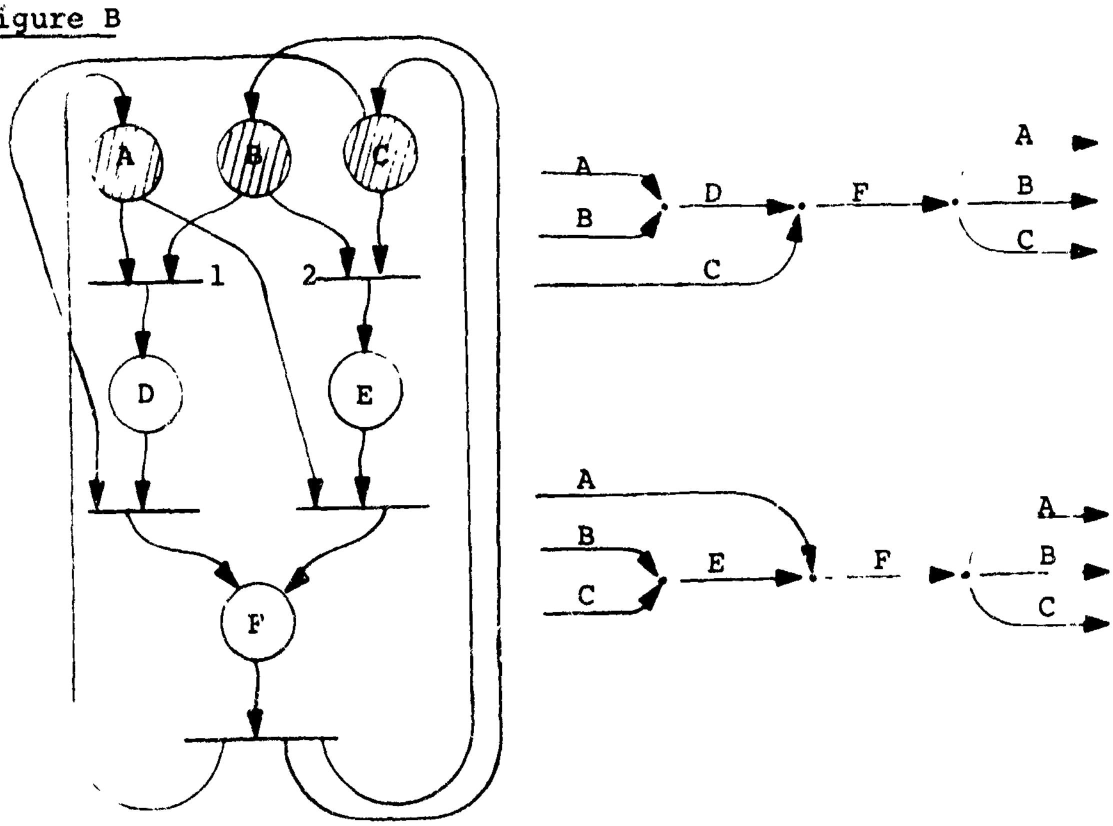
[1]For a comprehensive account of Petri nets we refe the reader to the "Final Report for the Information System Theory Project", RADC Contract # AF 30(602)-4211, by Dr. Anatol W. Holt et al.

Figure A



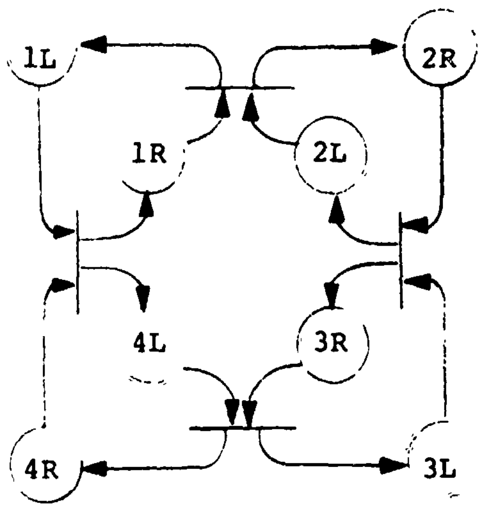
A net and an occurrence-graph representing its behavior. The shaded places are full. The broken lines represent time slices of the o-graph.

Figure B



A net with conflict and the o-cycles which constitute its basis. When A, B, and C are full, either transition 1 fires or transition 2 fires, but not both.

Figure C



1L : Ball 1 is moving counter-clockwise.

1R : Ball 1 is moving clockwise.

2L : Ball 2 is moving counter-clockwise.

etc.

In using Petri nets to describe a system, each place is associated with a proposition about the system. By interpretation, when a place is full, the proposition associated with it is true. In other words, the condition described by a proposition holds in the system when the associated place is full. The state of a system described by a given state of its net is the conjunction of the propositions associated with the full places.[2] Thus a net diagram together with a suitable initial assignment

[2]It is perhaps misleading to speak of "system states" here since a net does not necessarily define a totally ordered sequence of states. (Formally, this is because some transitions may fire concurrently - that is, their firings are not temporally ordered.) In this respect, nets differ fundamentally from state machines.

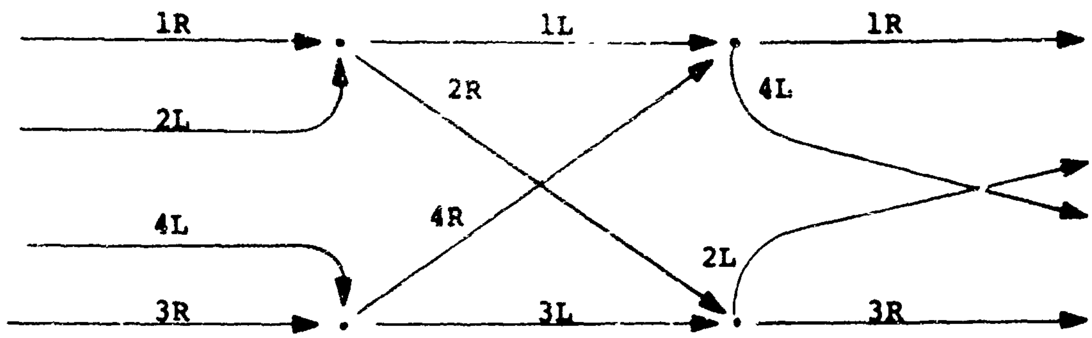
of place states (corresponding to the conditions which hold in the system initially) makes possible a formal simulation of the behavior of the corresponding system. Note that it is the occupancy of places which is viewed as having duration. Transitions merely bound places; the firing of a transition is not viewed as time-consuming -- rather, it is a separation of distinct place occupancies. Hence, the propositions associated with places describe conditions involving time-consuming operations or states. Figure C, for example, is a net representation of four balls moving and colliding on a single-lane circular track. The propositions describing the system are all of the form: "ball n is moving clockwise (or counter-clockwise)".

We may view an <u>occurrence-graph</u>, or <u>o-graph</u>, as a directed graph which represents a simulation history of some net. Formally, an o-graph consists of <u>vertices</u>, <u>arcs</u>, and <u>labels</u> associated with the arcs. Each label corresponds to some condition of the system being represented. (The words <u>label</u> and <u>condition</u> are therefore used interchangeably in this context.) Each arc represents an interval of place occupancy (or condition holding); the place (and hence the condition) is designated by the label associated with the arc. An inner vertex represents a transition firing and hence an <u>occurrence</u> in the system being represented. (The terms <u>inner vertex</u> and <u>occurrence</u> are accordingly

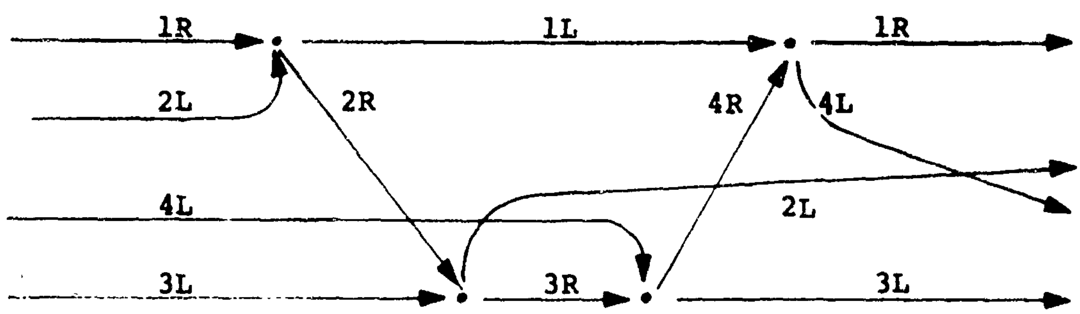
used interchangeably.) Thus an occurrence may be described as follows: the conditions of the input arcs cease to hold (the input places become empty); the conditions of the output arcs begin to hold (the output places become full). (See Figures A, B, and D for examples of o-graphs.)

Two occurrences are said to be temporally ordered if and only if there is a path from one to the other; the former precedes the latter. Note that some occurrence pairs in an o-graph are temporally ordered while others are not. Occurrences which are not ordered are said to be *concurrent*. Similarly, two arcs are temporally ordered if and only if there is a path from one to the other; arcs which are not temporally ordered are concurrent. A *time-slice* is a maximal set of pairwise concurrent arcs. A time-slice represents a possible state of the net (and hence of the system) during the history which the o-graph describes. (See Figure A.)

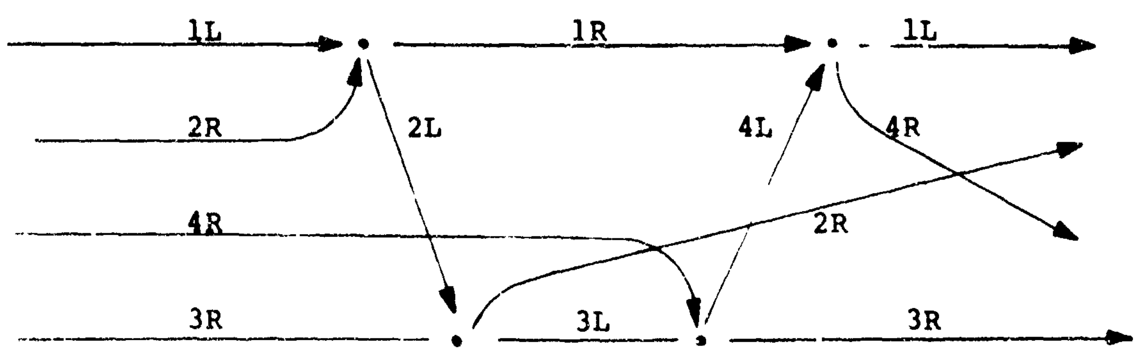
<u>Figure D</u>



(two balls moving clockwise and two counter-clockwise)



(three balls moving counter-clockwise and one clockwise)



(three balls moving clockwise and one counter-clockwise)

An o-graph may be decomposed at a time-slice. Two o-graphs may be composed if the terminal conditions of one are identical to the initial conditions of the other. An o-graph whose initial and terminal conditions are identical is termed an o-cycle. An o-graph formed by composing some number of copies of an o-cycle is termed a repetition stretch of the o-cycle. An o-cycle which cannot be decomposed into further o-cycles is termed an irreducible o-cycle. (The o-graphs shown in Figures A, B, and D are all irreducible o-cycles.) For every net together with a suitable assignment of place states, there is at least one basis, consisting of a finite set of irreducible o-cycles from which every possible simulation history may be generated by composition and decomposition. If the net contains no conflict, its basis consists of one irreducible o-cycle. Note that a given net diagram may be capable of several different disjoint behaviors given different initial place assignments. Figure D, for example, shows the bases for the three different behaviors of which the net in Figure C is capable.

BIBLIOGRAPHY

Armed Forces Technical Information Agency. "Automation of ASTIA 1960". December 1960, Reprinted February 1962. AD 247 000.

Auerbach Corporation. "Data Manager-1. Data Management System for a Time-Sharing Environment". Auerbach Corporation, Philadelphia, Pa. Research Report 7030, December 1967.

Buck, R. Creighton. "Studies in Information Storage and Retrieval on the Use of Gödel Indices in Coding". American Documentation, Vol. 12, No. 3, July 1961. pp. 165-171.

Burnaugh, H.P. "Data Base for the BOLD System". System Development Corporation, Santa Monica, California. TM-2306/001/02, August 1966.

Casey, Perry, et al. "Punched Cards. Their Applications to Science and Industry". Second edition. New York: Reinhold, 1958.

Conners, T.L. "ADAM - A Generalized Data Management System". The Mitre Corporation, Bedford, Mass. Proceedings of the AFIPS 1966 Spring Joint Computer Conference, Boston, Mass., April 1966. Washington, D.C.: Spartan, 1966.

Craig, J. and Goodroe, J. "General Design Specifications for a Random Access Storage Management System". Massachusetts Computer Associates, Wakefield, Mass. CA-6704-0422, April 1967.

"Data Management: File Organization". EDP Analyzer, December 1957.

Defense Documentation Center. "Evolution of the ASTIA Automated Search and Retrieval System". DDC, Washington, D.C. September 1963. AD 252 000.

Doudnikoff, Basil and Conner, A., Jr. "Statistical Vocabulary Construction and Vocabulary Control with Optical Coincidence". Jonker Business Machines, Inc., Washington, D.C. Statistical Association of Methods for Mechanized Documents, Symposium Proceedings. U.S. Dept. of Commerce, NLS, December 1965. pp. 177-180.

Drew, D.L. et al. "An On-Line Technical Library Reference Retrieval System". American Documentation, Vol. 17, No. 1, January 1966. pp.1-7.

Fossum, Earl G. et al. "Optimization and Standardization of Information Retrieval Language and Systems". Univac, Division of Sperry Rand, Blue Bell, Pa. July 1962.

Fossum, Earl G. and Kaskey, Gilbert. "Optimization and Standardization of Information Retrieval Language and Systems". Univac, Division of Sperry Rand, Blue Bell, Pa. January 1966. AD 630 797.

Franks, E.W. "The MADAM System: Data Management with a Small Computer". System Development Corporation, Santa Monica, California. September 1967.

General Electric Company. "Integrated Data Store. A New Concept in Data Management". Application Manual, General Electric Company, Information Systems Division, Bethesda, Md. July 1967.

General Electric Company. "Introduction to Integrated Data Store". General Electric Company, Computer Department, Phoenix, Arizona. April 1965.

Goldberg, J. et al. "Multiple Instantaneous Response File". Stanford Research Institute, Menlo Park, California. August 1961.

Holt, Anatol W. et al. "Information System Theory Project, The Nature of FFS: An Experiment in $\mathfrak{D}$-Theoretic Analysis". Applied Data Research, Inc., Princeton, N.J. March 1966.

Holt, Anatol W. at al. "Information System Theory Project: Vol. 1, $\mathfrak{D}$-Theory". Applied Data Research, Inc., Princeton, N.J. November 1965. AD 626 819.

Holt, Anatol W. et al. "Information System Theory Project, Final Report". Applied Data Research, Inc., Princeton, N.J. September 1968. AD 676 972.

Holt, Anatol W. "ISTP Edge-Notched Card System. (A Manual for the Information System Theory Project)". Applied Data Research, Inc., Princeton, N.J. February 1964.

Hubbell, Paul. "A Search for Improved Coding Methods for a Large-Scale Information Retrieval System". University of Pennsylvania, Moore School of Engineering. A Thesis. May 1964.

Jaster, J.J. et al. "The State of the Art of Coordinate Indexing". Defense Documentation Center, Washington, D.C. February 1962. AD 275 393.

Jonker, Frederick. "Indexing Theory, Indexing Methods, and Search Devices". New York: The Scarecrow Press, 1964.

Jonker, Frederick. "Design Considerations of Information Storage and Retrieval Machines". Documentation, Inc., Washington, D.C. April 1958.

Kennedy, F.L. and Brown, M.E. "The Applications of Computers to the APL Storage and Retrieval System". Johns Hopkins University, Applied Physics Laboratory, Bethesda, Md. TG-669. March 1965.

Kent, Allen. "Textbook on Mechanized Information Retrieval". Second edition. New York: Interscience, 1966.

Kochen, Manfred. "Toward Document Retrieval Theory -- Techniques for Document Retrieval Research, State of the Art -- On Natural Information Systems: Pragmatic Aspects of Information Retrieval". Aerospace Intelligence Data Systems, Appendices 1, 2, and 3. International Business Machines Corporation. 1963.

Kochen, Manfred et al. "High-Speed Document Perusal". International Business Machines Corporation. May 1962.

Kochen, Manfred. "Some Problems in Information Science". New York: The Scarecrow Press, 1965.

Koriagin, G.W. and Bunnow, L.R. "Mechanized Information Retrieval System for Douglas Aircraft Company, Inc.". Status Report, SM-39167. January 1962.

Landauer, Walter. "The Tree as a Stratagem for Automatic Information Handling". Graduate School of Arts and Sciences, University of Pennsylvania. A Thesis. 1962.

Lefkowitz, D. "Automatic Stratification of Descriptors". Moore School of Engineering, University of Pennsylvania. Report No. 64-03. September 1963.

Lefkowitz, D. and Prywes, N. "Automatic Stratification of Information". Proceedings of the AFIPS 1963 Spring Joint Computer Conference, Detroit, Michigan. May 1963. Baltimore, Md.: Spartan, 1963.

Lefkowitz, D. and Powers, R.V. "A List-Structured Chemical Information Retrieval System". In: Schecter, George (ed.). "Information Retrieval. A Critical View". Washington, D.C.: Thompson, 1967.

Leveille, Gilbert, et al. "A Simple and Versatile Punch Card System for Bibliographic Use". December 1963. AD 601 914.

Lin, Andrew D. "Key Addressing of Random Access Memories by Radix Transformation". Proceedings of the AFIPS 1963 Spring Joint Computer Conference, Detroit, Michigan. May 1963. Baltimore, Md.: Spartan, 1963.

Lowe, Thomas C. "Design Principles for an On-Line Information Retrieval System". Moore School of Engineering, University of Pennsylvania. December 1966.

Mathews, William D. "The TIP Retrieval System at M.I.T.". In: Schecter, George (ed.). "Information Retrieval. A Critical View". Washington, D.C.: Thompson, 1967.

Mooers, Calvin N. "The Application of Simple Pattern Inclusion Selection to Large-Scale Information Retrieval Systems". April 1959. AD 215 434.

Mooers, Calvin N. "Choice and Coding in Information Retrieval Systems". Zator Company. (No date given.)

Mooers, Calvin N. "Zatocoding for Punched Cards". Zator Company, Technical Bulletin #30.

Mooers, Calvin N. "Extensions of Pattern Inclusion Selection". 2TB-133. August 1959.

National Cash Register Company. "NCR 315 Electronic Data Processing System". MD 315-101 10-62. National Cash Register Company, Dayton, Ohio.

Olle, T.W. "INFOL: A Generalized Language for Information Storage and Retrieval Applications". In: Schecter, George (ed.). "Information Retrieval. A Critical View". Washington, D.C.: Thompson, 1967.

Orosz, G. and Takacs, L. "Some Probability Problems Concerning the Marking of Codes into the Superimposition Field". The Journal of Documentation, Vol. 12, No. 4, December 1956.

Pepinsky, Ray and Vand, Vladimir. "New Methods for Mega-Item Information Retrieval Using Small-Scale Machines". Report No. 48, The Groth Institute, 1960.

Peterson, W.W. "Addressing for Random Access Storage". IBM Journal of Research and Development, Vol. 1, No. 2, April 1957. pp.130-146.

Prywes, N. et al. "MULTILIST Organization of Storage and its Optimization -- Construction and Expansion of a Balanced Coding Tree -- Automatic Stratification of Information -- The Memory Synchronizer". Moore School of Engineering, University of Pennsylvania. "The MULTI-LIST System: Technical Report No. 1". Part 1, Volume 1. November 1961

Prywes, N.S. "Man-Computer Problem Solving with MULTI-LIST". IEEE Proceedings, December 1966.

Prywes, N.S. "The Organization of Files for Command and Control". University of Pennsylvania, Philadelphia, Pa. March 1964.

Savitt, Donald A. et al. "Association Storing Processor (ASP). The ASP Language - Formal Definition". Rome Air Development Center, Griffiss Air Force Base, New York. March 1967.

Schay, Geza, Jr., and Dauer, Francis W. "A Probabilistic Model of a Self-Organizing File System". SIAM Journal on Applied Mathematics, Vol 15, No. 4, July 1967. pp.874-888.

Schecter, George (ed.). "Information Retrieval. A Critical View". New York: Thompson, 1967.

Seidel, Mark. "Threaded Term Association Files". Statistical Association Methods for Mechanized Documents, Symposium Proceedings. U.S. Dept. of Commerce, NBS, December 15, 1965. pp.173-6.

Stiassny, S. "Mathematical Analysis of Various Superimposed Coding Methods". American Documentation, Vol. 11, No. 2, February 1960.

Taube, Mortimer. "Experiments with the IBM-9900 and a Discussion of an Improved COMAC as Suggested by these Discussions". April 1961.

Taube, Mortimer. "The Mechanization of Data Retrieval". In: "Studies in Coordinate Indexing". Volume 4. Washington, D.C. Documentation Inc., 1957.

Thomas J. Watson Research Center. "Computer Programming Techniques for Intelligence Analyst Application". Thomas J. Watson Research Center, Yorktown Heights, N.Y. Quarterly Report No. 2. October 1964.

Wong, E. "Time Estimation in Boolean Index Searching". International Business Machines Corporation, New York. December 1961.

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author) Applied Data Research, Inc. Corporate Research Center 450 Seventh Ave., New York, N.Y. 10001	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP N/A

3 REPORT TITLE

A Handbook on File Structuring

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)

Final Report

5 AUTHOR(S) (First name, middle initial, last name)

Robert M. Shapiro
Harry Saint
Robert E. Millstein
Anatol W. Holt
Stephen Warshall
Louis Sempliner

6 REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO OF REFS
September 1969	159	60

8a. CONTRACT OR GRANT NO F30602-69-C-0034 b. PROJECT NO 4594 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) CA-6908-2331 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) RADC TR-69-313, Vol I

10 DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11 SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

This report makes an initial attempt at presenting a coherent approach to the design and analysis of file structures. The relative efficiency of different file implementations is discussed as a function of usage statistics. The fundamental differences between item and descriptor-organized files are discussed in terms of input-output requirements. The report concludes with a discussion of batching, buffering and concurrency.

UNCLASSIFIED
Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
FILE STRUCTURING						
STORAGE/RETRIEVAL						